LPA MacPROLOG

LOGIC PROGRAMMING ASSOCIATES LIMITED (LPA)
END-USER LICENSING AGREEMENT

You (the Purchaser) have acquired a Package comprising Software and Documentation. By opening the disk enclosure you have agreed to abide by the terms of this Agreement, which are:-

1. Copyright
All copyright title and intellectual property rights in the Package are retained by LPA. LPA grants you a non-exclusive licence to use the package on a single machine only. You may make copies in any machine-readable or printed form for back-up or modification purposes in support of your own use on a single machine only (except insofar as any software is copy-protected) and to every copy or modification you must attach a permanent label giving the name of the Software and stating that it is the copyright of LPA. You may not copy the printed Documentation.

2. Multi-Computer Use
If you wish to use the Package on more than one machine you must inform LPA and pay a further licence fee depending on the number of computers to be used.

3. Transfer
You may transfer the Package and this single computer licence to another person if, and only if, (a) that person agrees to accept the terms and conditions of this agreement and (b) you transfer the whole of the Package and all copies and modifications (unless you destroy any copies and modifications which you retain) and (c) the copyright notice is attached to every copy and modification and (d) both you and the transferee sign and return the re-registration form and agreement contained in the reference manual. You may not transfer a multi-computer licence. **Unless all of these conditions are fulfilled you may not transfer the whole or any part of this package or this licence to any other person.** If you try to do so then your own licence is automatically terminated.

4. Updates
If you have completed the Registration Card (or, in the case of the Assignee, the re-registration card) and returned it to LPA, LPA may (but shall not be obliged to) sell you any update to the Package and this agreement shall apply also to any update and to any replacement disk. **Please note that any offers which LPA may make of updates at special rates are available only to the Registered Owner. It is therefore in your own interests to ensure that you are registered.**

5. Limited Warranty
LPA warrants only that the disk on which the software is provided is free from defects in materials and workmanship under normal use for 30 days from your receipt of it. LPA does not make any representation warranty or guarantee that the Package is fit for any particular purpose. If you return the Package and your receipt to LPA and can demonstrate to LPA that the disk has not been misused or used on defective or incompatible equipment but is nonetheless faulty then LPA will supply a replacement free of charge. Thereafter, if you have registered with LPA in accordance with paragraph 3 above LPA will, upon receipt by LPA of the original disk and LPA's current replacement fee, replace any copy-protected disk which has been corrupted. This is LPA's entire liability and your sole remedy. **In no event shall LPA be liable for any direct or consequential loss of any kind, except that which is unlawful to exclude.** If any such exclusion of liability shall be held to be unlawful for any reason then LPA's liability shall be limited to the one-time licence fee paid to LPA by the end-user upon the grant of this Licensing Agreement.

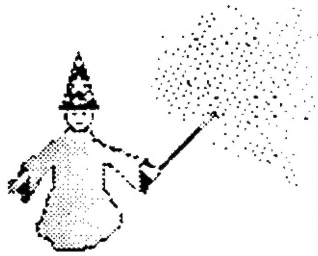
6. Term
You may terminate this Agreement at any time. LPA may terminate this Agreement if you are in breach of any of your obligations under this Agreement or if you become insolvent. Upon termination for any reason you undertake immediately to destroy the Package and any copy modification or merged portion in any form.

7. English Law
Unless you are resident in the United States of America (in which case this Agreement shall be governed by the law of the State of California) this Agreement is governed by English Law and the English Courts shall have sole jurisdiction in any dispute.

This is the complete statement of the entire terms of the Agreement between you and LPA. You acknowledge that you have read it and understood it and, that by opening the Package or permitting the Dealer to open it for you, you agree to be bound by its terms.

LPA MacPROLOG Reference Manual

Keith L Clark
Frank G McCabe
Nicky Johns
Clive Spenser



Logic Programming Associates Ltd
Studio 4
The Royal Victoria Patriotic Building
Trinity Road
London SW18 3SH

Read on ...

Introduction to LPA MacPROLOG

LPA MacPROLOG represents the successful combination of a state of the art Artificial Intelligence language and the user-friendly interface and enhanced graphics of the Apple Macintosh. By integrating PROLOG with the Macintosh philosophy of mice, menus and icons, and by implementing a true multi-window management system, LPA has provided PROLOG programmers with a sophisticated and rich programming development environment, previously unavailable on micro-computers.

LPA MacPROLOG uses an extension of the industry standard Edinburgh PROLOG syntax and an extended set of I/O and PROLOG data base primitives that greatly enhance its power as an AI language. The subset of *LPA MacPROLOG* comprising standard PROLOG primitives is compatible with Quintus PROLOG. In addition to these PROLOG orientated primitives, *LPA MacPROLOG* also provides a comprehensive set of primitives specific to the Macintosh environment of windows, menus, dialogues and graphics.

By virtue of its **fast incremental compiler,** *LPA MacPROLOG* claims not only to be the most advanced PROLOG system available on any micro, but also a powerful, general-purpose, high-level Macintosh application language which enables the **rapid prototyping** of Macintosh applications.

With its **powerful optimising compiler** and efficient implementation, *LPA MacPROLOG* also provides the basis for **quick final applications**. An example of this is the *LPA MacPROLOG* programming environment itself, which at nearly 400K of compiled PROLOG, is itself a testimony to the speed and power of the underlying language design and implementation.

In line with LPA's philosophy of high-level and declarative systems, *LPA MacPROLOG* provides access to the Macintosh Toolbox and QuickDraw routines in a simple but efficient manner and removes a lot of the pain of programming the Macintosh from the programmer. For example, there are powerful primitives to generate dialogues, install user menus, track the mouse, or draw and manipulate complex graphical objects.

There is a comprehensive range of pre-defined dialogues but in addition users can implement their own bespoke dialogues by describing the configuration of buttons, check boxes, edit fields, scrolling menus, icons, text and pictures by a list of terms. A single call then displays and reads from the dialogue. For the adventurous there are extended 'dynamic' dialogues where the dialogue configuration can be manipulated under the control of a running PROLOG program.

Menus can be simply installed, extended, removed or altered with a single call to a primitive. User selections on a menu automatically invoke the PROLOG programs associated with that menu item.

All the window handling routines used in the *LPA MacPROLOG* development environment are available to the programmer as primitives of the language. Windows can be created, displayed, hidden, moved. killed, or used as I/O channels through program calls. This allows applications to make full use of the multi-window management system of *LPA MacPROLOG* and also all the standard Macintosh facilities such as the clipboard and desk accessories.

A recent feature of *LPA MacPROLOG* is its unique high-level graphics system. where graphical objects are defined in a declarative and descriptive manner. This powerful combination of graphics and PROLOG appears within the programming environment in the shape of a call-graph option which enables users to graphically display and interact with programs. This concept of graphical debugging tools is one that LPA hopes to develop and expand on in the near future in line with users' wishes for a simpler and faster way of maintaining and refining programs.

Graphical applications are easily implemented by defining graphical tool programs which are then invoked when the user clicks on that tool's picture.

By combining *LPA MacPROLOG*'s window, menu, dialogue and graphics handling primitives, programmers can now develop serious complex applications and provide a simple and attractive user interface.

The latest addition to *LPA MacPROLOG* is an interface to both C and Pascal. This allows programmers to use code already developed and established in these languages from within PROLOG.

Future LPA plans include the implementation of colour, a database interface toolkit, and an expert system toolkit.

LPA MacPROLOG is the product of 8 years of LPA experience in the design and implementation of PROLOG systems. We trust you will find our time and effort worthwhile, and we are always happy to receive any comments you may have on *LPA MacPROLOG*, and news of how you're using it.

LPA MacPROLOG™ Reference Manual

Contents

Chapter 24 Graphic Description Language 228

Chapter 27 Advanced Drawing 320

Chapter 28 Tool Building 335

1 Edinburgh Syntax

The Edinburgh syntax of LPA MacPROLOG™ is essentially the syntax of the book

> *Programming in Prolog*
> by Clocksin & Mellish
> published by Springer-Verlag

It is the syntax which originated with the Edinburgh University DEC-10 compiler, hence its name.

In this Chapter we briefly summarise Edinburgh syntax. Those familiar with the syntax should pay special attention to sections 1.6 and 1.13, which describe some special features of MacPROLOG Edinburgh syntax.

Edinburgh terms

There are six types of Edinburgh term: *numbers*, *atoms*, *variable names*, *lists*, *strings* and *compound terms*.
Variable names and atoms are disjoint. A distinction is made between variable names and *variables*. Variable names are converted into variables when clauses are used or when terms are read in using the normal `read` primitive for Edinburgh terms. However, there are input primitives such as `gread` and `prompt_gread` that read in Edinburgh terms without converting variable names into variables. They will be left in the term as (quoted) atoms. These primitives also return the list of variable names in the read-in term as the value of an extra argument variable of the call.

1.1 Character set

MacPROLOG uses the 8-bit ASCII character set , including all the the special characters above ASCII 128 provided on the Mac. The characters are represented internally by 8-bit numbers (bytes) in the range 0..255.
(See the Appendices for the Macintosh character set.)

1.2 Separators and terminators

The normal term separator is the comma. This is used to separate terms in lists.
The full stop followed by a space or a carriage return is the normal term terminator, although in dialogue edit fields the terminating full stop is optional. If a full stop is not followed by a space or carriage return it is interpreted as a symbolic atom.
A space between an atom and a left round bracket (is also significant.
A space must also be used to separate an operator from an operand if they are the same token type, e.g. both alphanumeric tokens.

1.3 Comments

A comment can be inserted at any point in an Edinburgh program window. It is any sequence of characters that begins with /* and ends with */.

End of line comments (a sequence of characters started by a % character and continuing until the end of the line) are also supported.

1.4 Numbers

Numbers are either *integers* or *floating-point* numbers. An integer is any number with no fractional part.

A *positive integer* is written as a contiguous sequence of digit characters, with no leading sign character. A *negative integer* is a *sign* character (-) contiguously followed by a positive integer. A sign character not directly followed by a positive integer is not regarded as the sign of a number. Integers are stored as signed 24 bit numbers in MacPROLOG. The following are integers:

```
9821        211327        -32768        0
```

Floating point numbers are written in fairly conventional notation, as illustrated below:

```
2.34        10.3e99        12.003E-100        -0.81
```

One and only one period must be present in a floating point number, and it must be contiguously followed by a digit. As with an integer, a floating point number must start with a digit or a minus sign and a digit. It cannot start with a period. The e exponent is optional, but if used it must be contiguous to the number and must be contiguously followed by an integer. The following are not floating point numbers:

`.9`	/* starts with . */
`3e-22`	/* no . */
`34 e3`	/* space before e*/
`-.7e45`	/* no digit after -* /
`56e4.8`	/* exponent not an integer */
`91.`	/* no digit after . */

Floating point numbers may be written in fixed point notation (e.g. `123.45`) or scientific notation (e.g. `1.2345E2`). MacPROLOG will print numbers in the scientific notation if they cannot be printed in fixed point format (i.e. they are either too large or too small). This means in practice that all numbers with magnitudes between 1 and 99,999,999 inclusive are displayed without exponent, and any numbers with magnitude between 0.0000001 and 0.9999999 are displayed without exponent, provided that there are no digits after the 7th decimal place.

Thus	12345.678	is displayed as	`12345.678`
and	12345678	is displayed as	`12345678`
but	123456789	is displayed as	`1.2345678E8`
also	1.234567E-1	is displayed as	`0.1234567`
but	1.2345678E-1	is displayed as	`1.2345678E-1.`

Numbers have up to 8 digits of precision, and have an exponent in the range -127 to 127.

As with integers, negative floating point numbers have a sign character in front of them, and positive floating point numbers do not have a sign character. If you enter

```
+6.86
```

this will be interpreted as the operator + applied to the positive number `6.86`.

Note that MacPROLOG allows the free mixing of integers and floating point numbers. The arithmetic primitives automatically convert integers to floating point numbers where necessary; but there are also special primitives for integer only arithmetic.

1.5 Atoms

Atoms are of three types: *alphanumeric*, *symbolic* and *quoted*.

An *alphanumeric* atom is a lowercase letter (`a-z`) followed by a sequence of zero or more alphabetic characters or digits. The underscore character _ counts as an alphabetic character. The hyphen character - **does not**. Example:

```
apple   h45j   apple_cart
```

are three alphanumeric atoms.

A *symbolic* atom is a contiguous sequence of symbolic characters such as `*`, `>` or Δ. However the eleven symbols

```
( )  |  [  ]  ,  ' " { } %
```

cannot appear in a symbolic atom since they have a special syntactic role.

The full stop can appear in a symbolic atom, but remember that full stop followed by a space or carriage return is the term terminator.
For example:

```
&&/   <<   =3f   !   *>/*
```

are five symbolic atoms.
(Notice that the `/*` appearing in `*>/*` is part of the symbolic atom, and in this case is not interpreted as the start of a comment.)

A *quoted* atom is any sequence of characters surrounded by single quotes.
For example:

```
'Apple'   'The green ***'   '^hht'
```

are three quoted atoms.

The single quote character can be represented by two adjacent single quote characters.

To insert control codes use ~*letter* where **control-*letter*** is the control code.
For example, to insert the carriage return control character in a string, which is control-M (ASCII 13) use the pair of characters `~M`. The tilde, `~`, is an *escape* character. To insert a tilde use `~~`.

The maximum size of an atom is 255 characters (in the unquoted name).

1.6 Variable names

Variable names are alphanumeric sequences of characters beginning with an upper case letter (`A-Z`) or an underscore `_`.

```
Apple   _23   X
```

are three variable names. Quoting with single quotes over-rides the variable name convention.

```
'Apple'   'X'
```

are both quoted atoms.

An underscore on its own is an anonymous variable. Different occurrences of the `_` anonymous variable in a read-in term are mapped into different new variable names of the form `__n` where `n` is an integer. These names begin with a double underscore. To avoid a possible clash of names, you should not use double underscore variable names of this form in a term in which there are anonymous variables.

WARNING

In MacPROLOG, you **cannot** use quotes to over-ride the variable name convention if the quoted atom is also used unquoted in the same term *as a variable name*. For example, in the list

```
['X',X,'Y']
```

only the `'Y'` will be an atom. Both the quoted and unquoted occurrences of `X` will be interpreted as occurrences of the same variable name `X`.

1.7 Compound terms

A *compound term* is an atom or variable name immediately followed by a sequence of k terms enclosed in round brackets and separated one from the other by commas. It is of the form

$f(t_1, t_2, .., t_k) \qquad k \geq 1$

The f is the *functor*. The term t_i is the i'th argument. k is the *arity*.
When a variable name is used it is usually when the term is a condition of a clause and the variable name is a meta-variable standing for a predicate name. See the section on meta-variables below.

```
>(2,3)   likes(tom,mary)   read(X)
```

are three compound terms.

1.8 Lists

Lists are sequences of terms separated by commas and surrounded by square brackets. They are of the form:

$[t_1, t_2, \ldots, t_k]$ $\qquad k \geq 0$

When k=0 we have the empty list `[]`.

Example:

```
[23,apples,[X,7]]
```

is a list of three items, with its third element also a list.

1.8.1 List patterns

The form

$[t_1, t_2, \ldots, t_i \mid Variable]$ $\quad i \geq 1$

is the pattern of a list that begins with the i terms $t_1, t_2, \ldots, t_i$ *followed by* some remaining list of terms represented by the *Variable*.

An alternative to `|` is the sequence `,..` (a comma followed by two full stops). So the above pattern can be written

$[t_1, t_2, \ldots, t_i \;,..\, Variable]$ $\quad i \geq 1$

1.9 Strings

A string is a sequence of characters surrounded by double quote characters. It is an abbreviated notation for the list of decimal integer ASCII codes of the characters in the sequence.
For example.

```
"A boy"
```

is shorthand for the list of five integers

```
[65,32,98,111,121]
```

To insert the double quote character, use two adjacent double quote characters "".
To insert control codes use ~*letter* where **control-*letter*** is the control code.
For example, to insert the carriage return control character in a string, which is control-M (ASCII 13) use the pair of characters ~M. The tilde, ~, is an *escape* character. To insert a tilde use ~~.

1.10 Operators

As an alternative to the prefix form, compound terms of the form

`f(t)`

with arity 1 can be written in the alternative operator expression form

`f t`

providing `f` has been declared a *prefix* operator, or the operator expression form

`t f`

providing `f` has been declared a *postfix* operator. Compound terms of the form

$g(t_1, t_2)$

of arity 2 can be written in the alternative operator expression form

$t_1\ g\ t_2$

providing `g` has been declared an *infix* operator.

The declaration of an operator is achieved either by using the **Operators...** menu command or by using the `op` primitive (the operator declaration may be included in the '<LOAD>' program of an application - see the chapter on Implementing an Application).

An operator declaration is a term of the form

`op(priority,type,op_name)`

where `priority` is an integer between 1 and 1200, `op_name` is the operator name and `type` is one of:

`fx`	for non-associative prefix operator
`fy`	for right associative prefix operator
`xf`	for non-associative postfix operator
`yf`	for left associative postfix operator
`xfx`	for non-associative infix operator
`xfy`	for right associative infix operator
`yfx`	for left associative infix operator

When several operators have the same declaration, the `op_name` argument of the `op` declaration can be a list of operator names.

The `priority` determines the binding of an operator to its arguments when the argument of an operator is itself an operator expression. The lower the `priority` number, the more binding the operator.

As an example

```
2 * 5 + 8
```

is the compound term

```
+(*(2,5),8)
```

because * is a predeclared operator with priority 400 and – is predeclared with priority 500.

An operator name can be declared with more than one type.

Brackets can be used to over-ride the priorities. For example,

```
2*(5+8)
```

is the compound term

```
*(2,+(5,8))
```

The following are the predeclared operators of MacPROLOG Edinburgh syntax:

Priority	Type	Name
1200	xfx	:-
1200	xfx	-->
1200	fx	:-
1200	fx	?-
1150	fx	mode
1150	fx	public
1100	xf	?
1100	xfy	;
1100	xfy	|
1050	xfy	->
1000	xfy	,
900	fy	not
900	fy	spy
900	fy	nospy
700	xfx	=
700	xfx	\=
700	xfx	is
700	xfx	=..
700	xfx	\==
700	xfx	=:=
700	xfx	=\=
700	xfx	<
700	xfx	>
700	xfx	=<
700	xfx	>=
700	xfx	≥
700	xfx	≤
500	yfx	+
500	yfx	++
500	yfx	-
500	yfx	--

Priority	Type	Name
500	yfx	
500	yfx	
500	fx	-
500	fx	-
500	fx	
400	yfx	**
400	yfx	*
400	yfx	
400	yfx	
400	yfx	«
400	yfx	»
400	yfx	<<
400	yfx	>>
300	xfx	mod
200	xfy	^

1.11 Call Terms

A *call term* is an atom or a compound term that does not have `:-` as its functor.
Examples:

```
true
!
likes(tom,X)
append([],X,X)
```

`((likes(X,Y),male(Y))` is a conjunctive call term.

1.12 Clauses

A *clause* is a call term which is either an unconditional clause, or a term of the form:

t :- $t_1, t_2, \ldots, t_k$ $k \geq 1$

where `t` is a call term (the *head* of the clause) and each t_i is a variable name or a call term (the *conditions* of the clause).

A clause is the compound term

`:-(t,',' (t1,',' (t2,...',' (tk-1,tk)...)))`

where `:-` and `','` are functors. Notice the necessary quoting of the comma. `t` is the *head* of the clause and $t_1, t_2, \ldots, t_k$ is the *body*. The functor of `t` (or the name if it is an atom) is the relation that the clause is *about*.

Examples:

```
append([],X,X).
append([Hd|Tl],List,[Hd|ATl]):-append(Tl,List,ATl).

likes(tom,X):-horse(X),black(X).
```

The first two are clauses about `append`, and the third a clause about `likes`. MacPROLOG does not distinguish between different uses of the same relation name with different arities.

1.13 Meta-variables

There are two forms of meta-variable allowed in MacPROLOG: *predicate symbol* meta-variable and *condition* meta-variable.

If a body condition of a clause is a compound term with a variable name as the functor, i.e. if it is a condition of the form

```
Varname(t1,...,tn)
```

then `Varname` is a *predicate* meta-variable. The meta-variable must be bound to an atom which is the name of a defined relation by the time that the condition is evaluated, otherwise an error occurs.

For example, the following defines a `map` relation:

```
map(R, [], []).
map(R, [U|L], [MU|ML]):-
        l
        map(L,ML).
```

If a body condition of a clause is a variable name, this is a *condition* meta-variable. It must be bound to a term that represents a call by the time that it is evaluated. This term can be

(1) an atom, a call to a no argument relation
(2) a compound term of the form

```
relation_name(t1,t2,...,tn)
```

or (3) a list of the form

```
[relation_name,t1,t2,...,tn]
```

Both (2) and (3) represent calls to the program for `relation_name` with arguments `t1,t2,...,tn`.
(1) and (2) are normal call terms .
(3) is an additional way of representing a call which is allowed as the value of the call meta-variable.

For example, the following is the definition of the `not` primitive:

```
not(C):- C,!,fail.
not(C).
```

`not` can be called with an atom, a compound term or a list as argument.

These meta-variable facilities are somewhat more powerful than those usually allowed in Edinburgh syntax. Usually only the condition meta-variable is allowed, and that variable cannot be bound to a list to represent the call.

1.14 Definite Clause Grammars

Definite clause grammars allow the programmer to build programs which are intended for parsing applications, for example natural language parsers and programming language compilers. DCG rules are automatically converted into normal Prolog clauses which are then compiled in the usual way. In effect the DCG formalism is an extra syntactic layer on top of regular Prolog. The mechanism used to implement DCGs can also be used to provide alternative syntactic layers, for example the programmer may wish to have a macro facility, in which case this would be implemented in the same style as DCGs.

A DCG rule looks like a normal Prolog clause, but it has the particular form:

```
head --> body.
```

The meaning of this rule is that input which matches body can be reduced to head, or alternatively when using this rule in the conventional Prolog top-down style, "in order to recognise a head recognise a body".

The body is a sequence of terminals, non-terminals and conditions separated by commas. A *terminal* symbol is a literal which must appear in the input and is represented by a string or a list. For example if we were constructing an arithmetic expression parser we might have a rule which identified arithmetic operators:

```
add_op --> "+".
add_op --> "-".
mul_op --> [42]. % ASCII code for * is 42
mul_op --> [47]. % ASCII code for / is 47
```

A *non-terminal* takes the form of a normal compound term which may optionally include arguments. Arguments in non-terminals are useful to return the 'result' of the parse.
For example, when parsing an expression we might want the expression tree as the answer:

```
add_exp(Left+Right) --> add_exp(Left),"+",add_exp(Right).
add_exp(Left-Right) --> add_exp(Left),"-",add_exp(Right).
mul_exp(Left*Right) --> mul_exp(Left),"*",mul_exp(Right).
mul_exp(Left/Right) --> mul_exp(Left),"/",mul_exp(Right).
```

A more succinct way of writing this grammar would make use of variable functors:

```
add_op(+) --> "+".
add_op(-) --> "-".
mul_op(*) --> [42].
mul_op(/) --> [47].

add_exp(Op(L,R)) --> add_exp(L),add_op(Op),add_exp(R).
mul_exp(Op(L,R)) --> mul_exp(L),mul_op(Op),mul_exp(R).
```

There can be any number of rules for each definition of a non-terminal. The full arithmetic expression must allow both additive and multiplicative operators:

```
exp(E) --> add_exp(E).
exp(E) --> mul_exp(E).
```

Bodies of rules may also contain alternatives. These express alternative ways of obtaining a parse. Our two rules for exp can be expressed in the single rule:

```
exp(E) --> add_exp(E) | mul_exp(E).
```

Apart from terminal symbols and non-terminals a body of a DCG rule can contain *conditions*. Conditions are normal Prolog calls which are used to provide context sensitive tests although in fact any normal Prolog call can be used as a condition.

In our arithmetic expression parser we allow for arithmetic operators, but we must also allow for integers. We could write the rules for integers as a sequence of rules like:

```
int_exp(0) --> "0".
int_exp(1) --> "1".
        .
        .
int_exp(9) --> "9".
```

However, a neater solution involves the single rule:

```
int_exp(I) --> [C], {C=<57,          % 57 = ASCII for "9"
                     "0"=<C,
                      name(I, [C])}.
```

Note that in this rule we have both a condition and a terminal with a variable in it. The construction [C] is the standard way of picking up data from the input stream.

The complete arithmetic expression parser should allow for integer expressions, additive expressions, multiplicative expressions and bracketed expressions:

```
exp(E) --> mul_exp(E).

mul_exp(E) --> add_exp(E).
mul_exp(Op(L,R)) --> mul_exp(L),mul_op(Op),mul_exp(R).

add_exp(E) --> int_exp(E).
add_exp(Op(L,R)) --> add_exp(L),add_op(Op),add_exp(R).


int_exp(E) --> "(", exp(E), ")".
int_exp(I) --> [C], {C=<57,                 % 57 = ASCII for "9"
                     "0"=<C,
                      name(I,[C])}.

add_op(+)  --> "+".
add_op(-)  --> "-".
mul_op(*)  --> [42].          % ASCII code for * is 42
mul_op(/)  --> [47].          % ASCII code for / is 47
```

1.14.1 Using Definite Clause Grammars

A grammar built up through the use of DCGs is used to parse strings through the `phrase` primitive. This takes two or three arguments: a non-terminal, a string to parse, and an optional third argument which returns the remainder of the string after the parse.

We can use our arithmetic expression parser to parse and evaluate some simple arithmetic expressions:

```
phrase(exp(E),"2+3*(2-5)"),X is E?
X = -7
E = +(2,*(3,-(2,5)))
```

As with other Prolog programs grammars can be used to generate strings from non-terminals as well as parse strings. Furthermore, the strings might actually be lists of symbols rather than lists of bytes. A complete system might use one grammar to parse characters into tokens and another grammar to parse lists of tokens into full parse trees. A thrird grammar might be used in the final stages of a compiler to actually emit the target code of a compiled program.

1.14.2 Translation of DCG Rules to Clauses

DCG rules are translated into normal Prolog clauses by adding extra arguments to each of the non-terminals. Terminal symbols are translated to calls of the special built-in primitive '$C', and conditions are not translated. The extra arguments link the output of each terminal/non-terminal to the input of the next.

For example the rules for `add_op` are translated to:

```
add_op(+,S0,S1) :- '$C'(S0, 43, S1).
add_op(-,S0,S1) :- '$C'(S0, 45, S1).
```

The rules for `int_exp` are translated to:

```
int_exp(E,S0,S3):- '$C'(S0,40,S1),          % "("
                   exp(E,S1,S2),
                   '$C'(S2,41,S3).          % ")"
int_exp(I,S0,S1):- '$C'(S0,C,S1),           % [C]
                   C=<57,
                   "0"=<C,
                   name(I, [C]).
```

The translation of DCGs into Prolog clauses is performed whenever a file is consulted or when a window containing DCG rules is (re)compiled.

NOTE
The exact translation of DCGs into Prolog clauses varies with different Prolog systems, although the syntax of a DCG rule is standard. The programmer should not rely on the actual translation used.

For a more detailed introduction to DCGs the reader is recommended to read Clocksin & Mellish.

1.14.3 User Defined Term Expansion

DCGs are one example of a special pre-processor of Prolog text. The programmer may implement other alternative forms of pre-processing, for example a simple macro processor. This is implemented through the special `term_expansion` hook. If there is a definition of this program then for each clause that is `consulted` or compiled (though not directly `asserted`) `term_expansion` is called before actually compiling the clause.

The format of a `term_expansion` call is:

```
term_expansion(Input_term,Actual_clause)
```

The `Input_term` is the clause term as it is read during `consult` etc., and `Actual_clause` is the clause that will be compiled.

NOTE
If `term_expansion` is defined then the standard DCG processing **will not be invoked.**

1.15 Edinburgh Syntax Program Windows

An Edinburgh syntax program window comprises a sequence of clauses or DCG rules, each one terminated with a full stop. All the clauses *about* the same relation name must be contiguous and contained within one edit window, *even if* the head terms have different arities. A program window can contain the definitions for any number of different relations.

The LPA MacPROLOG™ Environment Guide contains a discussion on the recommended ways to use program windows for your source code.

2 Control

There are several control primitives for use in Edinburgh syntax programs:

the negation operator `not` or `\+`
the conjunction and disjunction operators `,` and `;`
the conditional operator `->`
the cut primitive `!`
`fail`, `true` and `repeat`
the meta-call primitive `call`
the generate and test primitive `forall`
the set constructors `findall`, `bagof` and `setof`
the general mapping utility `map`

Most of these primitives take call terms as arguments (see the Syntax chapter). If this call term is a conjunction, it should normally be enclosed in brackets.
For example,

```
not (X likes bob, X likes logic)
```

2.1 not - negation operator

```
     not call
or   \+ call                                (900 fy)
```

ARGUMENT

call : a call term

DECLARATIVE READING

call is false

USE

The negation as failure operator.
`not call` succeeds if and only if *call* fails.

It is defined by:

```
not X :- X,!,fail.
not X.
```

2.2 , - the conjunction operator

C1, C2 (1000 xfy)

DECLARATIVE READING

C1 and *C2* are true

ARGUMENTS

C1	: a call term
C2	: a call term

USE

Call *C1* is evaluated and then *C2* is evaluated. The conjunction succeeds if and only if both calls succeed.

2.3 ; - the disjunction operator

C1; C2 (1100 xfy)

ARGUMENTS

C1	: a call term
C2	: a call term

DECLARATIVE READING

C1 or *C2* is true

USE

The call succeeds if either *C1* or *C2* succeeds. *C1* is evaluated first. Only when all possible evaluation paths for *C1* have been explored will backtracking lead to an evaluation of *C2*.

A ! evaluated in the disjunction not only prevents backtracking within the disjunction, it also prevents backtracking to find alternative solutions to calls that precede the disjunction in the clause or query in which the disjunction appears.

A ! evaluated within the *C1* branch also prevents the use of *C2*.

2.4 -> - conditional evaluation

`test->call` (1050 xfy)

ARGUMENTS

`test` : a call term
`call` : a call term

USE

The -> call succeeds if and only if `test` succeeds and `call` succeeds. `call` can be a call to the disjunction operator `;`. In this case it is a conditional branch.

As with `;` a `!` evaluated in `test` or `call` not only has a local effect, but also the same effect as a `!` evaluated just before the -> call.

2.5 ! - backtracking control

`!`

USE

This is the backtracking control primitive. After it has been evaluated in a clause a backtrack to the `!` call will be interpreted as failure of the call `C` that invoked the clause. That is, it prevents the search for alternative solutions to any calls that precede the `!` in the body of the clause, and it will also prevent the use of other clauses to try to solve `C`.

Placed at the top level in a query conjunction it will prevent backtracking to find alternative solutions to calls that precede it in the query.

See the description of the logical operators `;` and -> for the effect of `!` inside these operators.

2.6 `fail` or `false` - force a failure

`fail`
`false`

USE

Does not match anything, call always fails.

2.7 `true` or `otherwise` - true

`true`
`otherwise`

USE

Call always succeeds.

2.8 **`repeat`** - backtracking loop entry

`repeat`

USE

Call always succeeds and repeatedly succeeds each time the evaluation backtracks to the call. It is defined as:

```
repeat.
repeat :- repeat.
```

2.9 **`call`** - evaluate argument as a call

`call(`*`term`*`)`

ARGUMENTS

term : a call term or a list of the form
[*relation_name*, t_1, ..t_n]

DECLARATIVE READING

term is true.

USE

It is equivalent to the call *term* (except where *term* is ! or includes ! - see below).
term can of course have as its functor any of the other logic operators including the conjunction operator.

A ! evaluated inside *term* only has a local effect. It only affects backtracking to find alternative solutions to calls that precede the ! within *term*.

2.10 `forall` - generate and test

`forall(`*`gen, test`*`)`

ARGUMENTS

gen	: call term
test	: call term

DECLARATIVE READING

For all the solutions of *gen*, `test` is true.

USE

The `forall` primitive is a high level concept that can often replace recursion in a program.

The following program defines a 'prime number' using `forall`. It can be used to check if a number is prime. A positive prime number `X` is a number that is not exactly divisible by any integer `Y`, where `Y` is in the range $2 \le Y < X$. This definition is formalised as:

```
prime(X) :-
  forall(in_range(2,X,Y), not divides(X,Y)).
```

`in_range` and `divides` are defined as:

```
in_range(Startval,Endval,Startval).

in_range(Startval,Endval,Val) :-
     Newstart is Start ++ 1,
     Newstart < Endval,
     in_range(Newstart,Endval,Val).

divides(Val1,Val2) :-
  ÷(Val1,Val2,Divisor),        % or Divisor is Val1 ÷ Val2
  integer(Divisor).
```

This is not a very efficient program for checking prime numbers, but it is a correct one.

DEFINITION

```
forall(A,B) :-
  not call((A,not B)).
```

After the evaluation of a `forall` all variables in *gen* and `test` are left unbound.

2.11 **`findall`** - construct a list of all the found solutions of a call

`findall(`*`term`*`, `*`call`*`, `*`list`*`)`

ARGUMENTS

`term`	: any term
`call`	: any call term
`list`	: a variable or list pattern

USE

`list` will be unified with a list of instantiations of *`term`*, one for each successful evaluation of *`call`*. The instantiations of *`term`* correspond to the different solution bindings for the variables in *`call`*. At the end of the evaluation no variable in *`call`* will be bound. All the 'local' variables in *`call`* , the variables which do not appear in *`term`*, or in any other condition in the clause or query in which the `findall` is used, are implicitly existentially quantified.

For correct use, all 'global' variables of *`call`*, variables that are also used in other conditions in the clause or query, should be bound to variable free terms before the `findall` is evaluated.

Example use:

```
male(P),findall(X,(gives(P,Y,X),female(Y)),L)
```

binds `L` to the list of all the things `X` that are given by male `P` to some female `Y`. The `Y` is local to the `findall`, the `P` is global.

PRAGMATICS

This is faster than `bagof` (see below) and should be used instead of `bagof` when there will be no unbound global variables in the *`call`* for which solutions are to be found.

2.12 bagof - find the list of solutions to a query for different bindings of its global variables

bagof(*eterm*, *existential_call*, *list*)

ARGUMENTS

list	: variable, will be bound to a list of terms
eterm	: term
existential_call	: term which is of the form *v1^v2^..^vk^call* , k≥0 where *call* is a call term and *v1, .., vk* are variables in *call*. When k=0 just the *call* is given.

USE

At the time of the call, *call* will generally contain variables. The variables in *call* that are not in either *eterm* or the sequence *v1, .., vk* of variables preceding *call* are the global variables of the bagof call. *v1, .., vk* are the existentially quantified variables of *call*.

bagof partitions the list of all the values of *eterm* for all the solutions of *call* by different solution values for these global variables.

That is, suppose that in the space of all the successful evaluations of *call* there are *n* different sets of bindings for its global variables. Then bagof will backtrack giving *n* differerent answers. Each answer will comprise a set of bindings for the global variables, and a corresponding value for *list* which comprises the instances of *eterm* for the different solutions of *call* that make this assignment to the global variables.

For example, suppose that we have the following definitions of gives and male:

```
gives(keith,sue,flowers)
gives(keith,sue,earrings)
gives(keith,bob,pen)
gives(sue,keith,guinness)
gives(bob,sue,ink)
gives(bob,keith,ball)
male(keith)
male(bob)
```

The call:

```
bagof(Person,Present^(gives(Giver,Person,Present),male(Giver)),L)
```

can be read as:

L is the list of Person(s) such that
male Giver gives Person *some* Present.

Its evaluation will generate two different answers:

```
L = [sue,sue,bob], Giver = keith
L = [sue,keith],   Giver = bob
```

Notice that sue appears on the first answer list twice because it appears in two different solutions of the two conditions paired with the same value keith of the global variable Giver. To obtain only one occurrence of sue we need to use setof described below.

2.13 **setof** - find the set of solutions to a query for different bindings of its global variables

setof(*eterm*, *existential_call*, *list*)

ARGUMENTS

list	: variable, will be bound to a list of terms
eterm	: term
existential_call	: term which is of the form *v1^v2^..^vk^call* , $k \geq 0$ where *call* is a call term and *v1, .., vk* are variables in *call*. When k=0 just the *call* is given.

USE

Exactly the same form of use as `bagof` described above. The difference is that the bindings for `list` will be ordered lists of terms without duplicates. The terms are ordered by the `@<` primitive as a list of increasing terms.

Thus, for the `gives` and `male` definitions of the `bagof` example, the two anwers to the call

```
setof(Person,Present^(gives(Giver,Person,Present),male(Giver)),L)
```

will generate two different answers:

```
L = [bob,sue],   Giver = keith
L = [keith,sue], Giver = bob
```

2.14 map - map a relation over a list

```
map(rel,inlist)
map(rel,inlist,outlist)
map(rel,inlist,invalue,outvalue)
map(rel,inlist,outlist,invalue,outvalue)
```

ARGUMENTS

rel	: relation name or unary compound term
inlist	: list
outlist	: variable or list
invalue	: term
outvalue	: variable

USES

For each use of `map`, the term *rel* may be either a relation name or a unary compound term of the form

reln(arg)

whose principal functor *reln* is a relation name.

1. Two argument use. If *rel* is a unary relation name, this call will test if all the elements of the list *inlist* satisfy *rel*.
For example.

```
map(integer,[1,2,-5,89])
```

succeeds.

If *rel* is a unary compound term of the form *reln(arg)* then the call tests if all the elements *el* of *inlist* satisfy *reln(arg,el)*.
For example,

```
map(>(10), [3,2,9,-3,1])
```

succeeds since all the numbers in the given list are less than 10.

2. Three argument use. If *inlist* is given, `map` will produce a list *outlist* such that each element of *outlist* is in the relation *rel* to the corresponding element of *inlist*. Alternatively, `map` may be used or to check that a given *outlist* is in this relationship to *inlist*.

For example, the call

```
map(charof,[a,b,c],List)
```

will bind `List` to the list `[97,98,99]`.

The call

```
map(*(8),[1,2,3,4],List)
```

will bind `List` to `[8,16,24,32]`.

3. Four argument use. The given `rel` is applied to 'accumulate' the elements of `inlist` using `invalue` as the initial value of the cumulated term. The variable `outvalue` is bound to the final cumulated term.

For example, the call:

```
map(++,[1,2,3,4,5],0,Val)
```

will bind `Val` to the sum of 1,2,3,4 and 5.

4. Five argument use. The given `rel` is applied both to produce an `outlist` from `inlist`. and to 'accumulate' the elements of `inlist` using `invalue` as the initial value of the cumulated term, binding the variable `outvalue` to the final cumulated term.

For example, the call

```
map(ts(8), [1,2,3,4], L, 0, Val)
```

where the definition of `ts` is

```
ts(N, E1, E2, V1, V2):-
        E2 is E1 * N,
        V2 is V1 ++ E2.
```

will bind `L` to `[8,16,24,32]` and `Val` to `80`.

3 Type Primitives

There are seven type checking primitives: `var`, `nonvar`, `atom`, `atomic`, `integer`, `number` and `float`.

3.1 `var` - test for variable

`var(arg)`

ARGUMENTS

arg : any term

DECLARATIVE READING

arg is an unbound variable. The call fails for any other type of argument.

3.2 `nonvar` - test for not being a variable

`nonvar(arg)`

ARGUMENTS

arg : any term

DECLARATIVE READING

arg is not an unbound variable. The call fails if *arg* is an unbound variable.

3.3 `atom` - test for atom

`atom(arg)`

ARGUMENTS

arg : any term

DECLARATIVE READING

arg is an atom. The call fails for any other type of argument.

3.4 `atomic` - test for atom or number

`atomic(`*`arg`*`)`

ARGUMENTS

arg : any term

DECLARATIVE READING

arg is an atom or a number. The call fails for any other type of argument.

3.5 `integer` - test for integer

`integer(`*`arg`*`)`

ARGUMENTS

arg : any term

DECLARATIVE READING

arg is an integer. The call fails for any other type of argument.

3.6 `number` - test for number

`number(`*`arg`*`)`

ARGUMENTS

arg : any term

DECLARATIVE READING

arg is a number - an integer or a floating point number. The call fails for any other type of argument.

3.7 `float` - test for non integer number

`float(`*`arg`*`)`

ARGUMENTS

arg : any term

DECLARATIVE READING

arg is a non-integer number. It has a fraction part. The call fails for any other type of argument.

4 Metalogical Primitives

In addition to the usual `arg`, `functor` and `=..` term manipulation primitives, MacPROLOG has two primitives `tohollow` and `toground` for converting between ground terms and hollow terms.

In a *ground term*, variable positions are represented by variable name atoms, in a *hollow term* they are represented by variables. The variables of a hollow term can be instantiated by unification, the variable names of a ground term will only unify with exactly the same variable name atoms.

The `tohollow` primitive can be used to convert into hollow terms the ground terms that are read in by the various ground read primitives, e.g. `gread`. The `toground` primitive can be used to ground a hollow term prior to pretty printing or some other recursive traversal where it is convenient not to have to deal with the case of unbound variables. It copies the hollow term replacing variables by underscore variable names or any given bindings.

The standard Edinburgh `numbervars` is also available. This grounds the variables of a term without copying. The primitive `varsin` finds all the variables in a term.

4.1 **`arg`** - argument selector

`arg(`*`N, T, arg`*`)`

ARGUMENTS

N	: integer
T	: non-variable term
arg	: any term

DECLARATIVE READING

arg is the *N*'th argument of term *T*.

USE

arg is unified with the *N*'th argument of *T*. It can be used to retrieve or check the argument value.

Example use:

```
arg(2,app([2],[3,4],Z),Arg)
```

binds `Arg` to `[3,4]`

4.2 `functor` - term decomposition/construction

`functor(T, F, N)`

ARGUMENTS

T	: any term
F	: atom or variable
N	: integer or variable

DECLARATIVE READING

T is a term with functor *F* and arity *N*.

USES

If *T* is a non-variable term, then the primitive finds the functor and arity of *T* and unifies their respective values with *F* and *N*.
The functor of a list is . and its arity is 2.
The functor of an atom or number is its value and its arity is 0.

For example:

```
functor(app(X,[1],Z),F,N)
```

binds `F` to `app` and `N` to `3`.

If *T* is a variable, then *F* and *N* must be an atom and integer respectively. The primitive binds *T* to a term with functor *F* with *N* new and different variables as arguments.

For example:

```
functor(T,likes,2)
```

binds `T` to `likes(_1,_2)`.

4.3 =.. - list to term conversion

term* =.. *list

ARGUMENTS

term	: compound term or variable (if *list* not a variable)
list	: list or variable (if *term* not a variable)

DECLARATIVE READING

list is a list comprising the functor of *term* followed by a list of its arguments. Traditionally this primitive is pronounced 'univ' after the same primitive in the original PROLOG implemented in Marseille in 1972.

USES

If *term* is a variable, a compound term is constructed from *list*.
If *list* is a variable, a list is constructed from *term*.
If both are non-variables, the list form of *term* is constructed and unified with *list*.

Example uses:

```
f(X,7) =.. Y
```

binds `Y` to `[f,X,7]`

```
Z =.. [likes,X,mary]
```

binds `Z` to `likes(X,mary)`

```
f(X,7) =.. [F,3,Y]
```

binds `F` to `f`, `X` to 3 and `Y` to 7

PRAGMATICS

Because MacPROLOG has more powerful meta-call facilities than other Edinburgh syntax systems there is no need to use this primitive for the construction of calls. The availability of the relation name meta-variable means that the relation name of a call can be changed without the use of this primitive and the fact that a meta-variable can be bound to a list at the time of the call means that it is not needed even when the number of arguments is also changed. Consult the section on meta-variables in the Syntax chapter.

However, if you want to preserve portability, you should continue to use this primitive to construct calls. If you are only developing for use with MacPROLOG it is more efficient if you avoid its use.

4.4 **varsin** - find all the variables in a term

varsin(*term*, *varlist*)

ARGUMENTS

term	: any term
varlist	: variable, will be bound to a list of all the variables in *term*

USE

To find the variables in a term.

4.5 **tohollow** - convert a ground term to a hollow term

tohollow(*ground_term*, *hollow_term*, *varnames*)
tohollow(*ground_term*, *hollow_term*, *varnames*, *vars*)

ARGUMENTS

ground_term	: variable free term
hollow_term	: variable, will become hollow copy of *ground_term*
varnames	: variable, or a list of atoms
vars	: variable, will be bound to the list of variables in *hollow_term* that have replaced the variable names in *ground_term*

USES

1. <u>Three argument use, *varnames* a list of atoms.</u> Only the atoms in the *varnames* list of atoms are replaced by variables in the *hollow_term* copy of *ground_term*. Any atoms can be on the *varnames* list. For example, the call

```
tohollow(likes(keith,'Person'):-
                 likes('Person','PROLOG'),
                 Hollow,['Person'])
```

will bind `Hollow` to

```
likes(keith,_1):-likes(_1,'PROLOG')
```

where `_1` is a variable but `'PROLOG'` is still an atom.

The main use of this form of the call is to convert a term and its associated list of variable names that has been read in using `prompt_gread` or `gread`, or retrieved from the data base using the two argument form of `clause`, into a hollow term that can be used in an evaluation.

2. <u>Four argument use.</u> Essentially the same as the above three argument use except that the list of variables in *hollow_term* that have replaced the atoms on *varnames* is also returned. The *i-th* variable on *vars* is the replacement for the *i-th* variable name on *varnames*.
This is useful if you want to subsequently output bindings for these variables using the original variable names.

4.6 **togroand - convert a hollow term to a ground term**

```
toground(hollow_term,ground_term)
toground(hollow_term,ground_term,varnames)
toground(hollow_term,ground_term,vars,varnames)
toground(hollow_term,ground_term,vars,
                varnames,usednames)
```

ARGUMENTS

hollow_term	: any term
ground_term	: a variable, will become *hollow_term* with all the variables replaced by unique underscore variable names
varnames	: variable, or list of atoms
vars	: variable, or list of variables/terms
usednames	: variable, will be bound to the list of all the variable names in *ground_term* that have replaced variables in *hollow_term*

USE

1. Two argument use. Simple use to convert any term into a variable free term. *ground_term* is bound to a copy of *hollow_term* in which all the variables are replaced by new atoms beginning with underscore. Different variables are replaced by different underscore names, and there will be no clash with any atom that begins with underscore which already appears in *hollow_term*.

2. Three argument use - *varnames* a variable. *varnames* will be bound to the list of all the underscore variable names that have been used to replace variables in *hollow_term*.

3. Four argument use - *varnames* and *vars* both variables. Same as the three argument use except that the corresponding list of the replaced variables is also returned as the binding for *vars*.

4. Five argument use - *varnames* a list of atoms. *vars* a list of variables appearing in *hollow_term*, *usednames* a variable.

The *varnames* list of atoms will be used to replace the variables of *vars* in *hollow_term* in the order in which they are given - the first variable of *vars* is replaced by the first atom on the *varnames* list, and so on. *vars* and *varnames* must be the same length.

If there are more variables in *hollow_term* than given in *vars*, new underscore names are given to the extra variables.

More generally, the elements of *vars* can be any terms. Non-variable terms on *vars*, and the corresponding atoms on *varnames* are ignored - but the lists must still be the same length. This relaxation allows lists of names and variables returned by some prior use of `tohollow` to be used as arguments to `toground` without having to remove variables that may have been bound by the intervening evaluation.

Finally, *usednames* is bound to the complete list of the actual variable names that have been used to replace variables in *hollow_term*.

4.7 **`numbervars`** - number the variables in a term

`numbervars(`*`term, start, next`*`)`

ARGUMENTS

`term`	: term
`start`	: integer
`next`	: variable

USE

The variables in the term *`term`* are instantiated to terms of the form

`'$VAR'(`*`n`*`)`

where *n* is an integer. The first value of *n* to be used is given by the *`start`* argument. The next distinct variable of *`term`* will be `'$VAR'(`*`n+1`*`)`, and so on.

When all the variables of *`term`* have been thus instantiated, the *`next`* argument will be bound to the next value of *n* that would be used. i.e. the last variable of *`term`* has the form `'$VAR'(`*`next-1`*`)`.

4.8 **`phrase`** - test if list can be parsed as a phrase of some type

`phrase(`*`type, list`*`)`
`phrase(`*`type, list, remlist`*`)`

ARGUMENTS

`type`	: non-terminal symbol
`list`	: list of terms
`remlist`	: remaining list of terms

DECLARATIVE READING

The sequence of terms in *`list`* can be parsed into a phrase of the given *`type`*.
In the three argument form, the first part of the sequence of terms in *`list`* can be parsed into a phrase of the given *`type`*, leaving the sequence of terms in *`remlist`*.

(See the section on Definite Clause Grammars in the Syntax chapter.)

5 Standard Arithmetic

MacPROLOG supports both integer and floating point arithmetic. There are four primitives ++ -- ** and // that require integer arguments. All the other primitives will take either integer or floating point arguments. Automatic conversion between integers and floating point numbers takes place wherever necessary.

This chapter describes the four standard arithmetic operations, and the arithmetic expression primitives `is`, =:= and =/=. Number comparison primitives are also included here.

The trigonometric functions and bit manipulation primitives are described in the Extended Arithmetic chapter.

Possible errors

Where an arithmetic operation results in arithmetic error, MacPROLOG signals an "Arithmetic error"(error 1). You will get the error when overflow or underflow occurs, or if you attempt an operation with an undefined result such as dividing by 0 or taking the square root of a negative number. The error will cause a run time error handling package to be invoked which displays the offending call with a special atom in place of the variable argument of the call. The atom indicates the type of error. For example, ∞ indicates arithmetic overflow and -∞ arithmetic underflow. All the other atoms begin with the letters **NAN** standing for Not A Number.

Accuracy

MacPROLOG uses 80 bits of accuracy in its floating point arithmetic calculations.

The integer arithmetic primitives use 24 bits of accuracy.

5.1 `is` - expression evaluator

`exp1 is exp2` (`700 xfx`)

ARGUMENTS

`exp1 exp2` : arithmetic expressions

DECLARATIVE READING

`is` should be read as an equality test that its two expression arguments have the same value.

USES

(1) When one of the arguments is a variable *v* the other argument is evaluated and *v* is bound to its value, e.g.

```
X is 7 + 89 * sin(Y)
```

(2) When both arguments are non-variable expressions they are both evaluated and their values tested for equality.

EXPRESSION SYNTAX

An *arithmetic expression* is

a *variable*
or a *number* (or a list of one number: `[n]` is equivalent to *n*)
or a term which is constructed using the variables, numbers and the predeclared operators:

```
op(300, xfx, [mod]).
op(500, yfx [+,-,\/,/\]).
op(500,fx [+,-,\,++,--]).
op(400,yfx,[*,**,/,//,«,»,<<,>>]).
```

and terms of the form

`rel(exp1,exp2)`

such that a call of the form

`rel(exp1,exp2,val)` with *`val`* a variable

will bind *`val`* to a numeric value.

The list of one number `[n]` evaluates to *n*. It means that a string of one character `"c"` (which is a list of one number : the ASCII code for `c`) is equivalent to the ASCII code for `c` in an arithmetic expression. This equivalence is particularly useful when using the `put` and `skip` primitives (see Chapter 11), both of which can have arithmetic expression arguments.

5.2 + addition

+(num1,num2,num3) (500 yfx)

ARGUMENTS

num1	: number
num2	: number
num3	: variable

DECLARATIVE READING

num1 + *num2* = *num3*

USE

Only one three argument use of + is permitted: where we are adding two numbers together to produce a result number. An error will occur if these conditions are not met.

5.3 - subtraction or minus operation

-(num1,num2,num3) (500 yfx)
-(num1,num3) (500 fx)

ARGUMENTS

num1	: number
num2	: number
num3	: variable

DECLARATIVE READING

num1 - *num2* = *num3* or -*num1* = *num3*

USES

Only one three argument use of - is permitted: where we are subtracting two numbers to produce a result number.

There is only one two argument use: when the negative of a given number is to be returned. An error will occur for any other attempted use.

5.4 * multiplication

`*(num1,num2,num3)` (400 yfx)

ARGUMENTS

num1	: number
num2	: number
num3	: variable

DECLARATIVE READING

num1 * *num2* = *num3*

USE

Only one use of * is permitted: where we are multiplying two numbers together to produce a result number. An error will occur if these conditions are not met.

5.5 ÷ or / division

`÷(num1,num2,num3)` (400 yfx)
`/(num1,num2,num3)` (400 yfx)

ARGUMENTS

num1	: number
num2	: number
num3	: variable

DECLARATIVE READING

num1 ÷ *num2* = *num3*

USE

Only one use of ÷ is permitted: where we are dividing one number by another to produce a quotient number. An error will occur if these conditions are not met.

5.6 ++ integer addition

`++(num1,num2,num3)` (500 yfx)

ARGUMENTS

num1	: number
num2	: number
num3	: variable

DECLARATIVE READING

num1 + *num2* = *num3*

USE

Only one use of ++ is permitted: when we are adding one integer to another. An error will occur if *num1* and *num2* are not integers.

PRAGMATICS

The integer forms of +, −, * and ÷ are supplied for increased efficiency. Use them when you know that all the numbers involved will be integers.

5.7 -- integer subtraction

`--(num1,num2,num3)` (500 yfx)

ARGUMENTS

num1	: number
num2	: number
num3	: variable

DECLARATIVE READING

num1 - *num2* = *num3*

USE

Only one use of -- is permitted: where we are subtracting one integer from another. An error will occur if *num1* and *num2* are not integers.

5.8 ** integer multiplication

`**(num1,num2,num3)` (400 yfx)

ARGUMENTS

num1	: number
num2	: number
num3	: variable

DECLARATIVE READING

num1 * *num2* = *num3*

USE

Only one use of `**` is permitted: where we are multiplying one integer by another. An error will occur if *num1* and *num2* are not integers.

5.9 // integer division

`//(num,divisor,quotient)` (400 yfx)

ARGUMENTS

num	: integer
`divisor`	: integer
`quotient`	: variable

DECLARATIVE READING

`quotient` is the integer part of the result of dividing *num* by `divisor`

USE

Only one use, to find the whole number of times one integer divides another. An error will occur if *num* and `divisor` are not integers.

For example, the call

```
//(38,7,X)
```

will bind X to 5.

5.10 mod - the modulus of one number relative to another

mod(*num, div, rem*) (300 xfx)

ARGUMENTS

num	: a number
div	: a number
rem	: a variable

DECLARATIVE READING

num = *div* * Q + *rem* , *rem* ≤ *div* and Q an integer

USE

Only one use, to find the modulus of given *num* relative to *div*.

For example, the call

```
mod(17,5,R)
```

will bind R to 2.

5.11 =:= - expression equality test

exp1* =:= *exp2 (700 xfx)

ARGUMENTS

exp1 exp2 : arithmetic expressions, see the description of is

USE

To evaluate *exp1* and *exp2* and confirm that they have the same value. This operator is redundant since this operation is also performed by the MacPROLOG is. It is included for compatibility with other Edinburgh syntax systems.

5.12 =\= - expression inequality test

exp1* =\= *exp2 (700 xfx)

ARGUMENTS

exp1 exp2 : arithmetic expressions, see the description of `is`

USE

To evaluate *exp1* and *exp2* and confirm that they do not have the same value. It is equivalent to

```
not exp1 =:= exp2.
```

5.13 < - check if a number is less than another number

number1* < *number2 (700 xfx)

ARGUMENTS

number1 : number
number2 : number

DECLARATIVE READING

number1 < *number2*

The first number is less than the second number.

USE

Checking only. < must be called with both arguments given. In this case the call succeeds if the first number is numerically less than the second number.

For example, `2<3` is true, as is `-1<1.32`.
But neither `10<9` nor `4.4<4.4` is true.

5.14 > - check if a number is greater than another number

number1* > *number2 (700 xfx)

ARGUMENTS

number1	: number
number2	: number

DECLARATIVE READING

number1 > *number2*

The first number is greater than the second number.

USE

Checking only. > must be called with both arguments given. In this case the call succeeds if the first number is numerically greater than the second number.

5.15 ≥ or >= - check if a number is greater than or equal to another number

number1* ≥ *number2 (700 xfx)
or ***number1* >= *number2*** (700 xfx)

ARGUMENTS

number1	: number
number2	: number

DECLARATIVE READING

number1 ≥ *number2*

The first number is greater than or equal to the second number.

USE

Checking only. ≥ must be called with both arguments given.

5.16 ≤ or =< - check if a number is less than or equal to another number

```
   number1  ≤ number2          (700 xfx)
or number1  =< number2         (700 xfx)
```

ARGUMENTS

number1	: number
number2	: number

DECLARATIVE READING

number1 ≤ *number2*

USE

Checking only. ≤ must be called with both arguments given.

6 Extended Arithmetic

6.1 sqrt - the square root of a number

sqrt(*num*, *root*)

ARGUMENTS

num	: number or variable (provided *root* is a number)
root	: number or variable (provided *num* is a number)

DECLARATIVE READING

num = *root* * *root*

USES

1. Taking a square root. If *num* is a number and *root* is a variable then the *root* argument is bound to the square root of *num*. For example, the call

```
sqrt(9,X)
```

results in X being bound to 3. The *num* argument should be positive, otherwise the "Not a number error" (1) is signalled.

2. Squaring a number. The sqrt primitive can be used to square a number. If the *root* argument is a number and the *num* argument is a variable, *num* will be bound to *root* squared.

3. Checking. If both arguments are numbers at the time of the call then the *num* argument is checked to see that it is *root* squared.

6.2 abs - the absolute value of a number.

abs(*num*, *abs*)

ARGUMENTS

num	: number
abs	: variable

DECLARATIVE READING

| *num* | = *abs*

USE

Only one use of abs is supported: to take the absolute value of a number.
For example, the call

```
abs(-3,A)
```

will bind A to 3.

6.3 `int` - truncate a number towards zero

`int(num, int)`

ARGUMENTS

num	: number
int	: variable or integer

DECLARATIVE READING

int is the nearest integer to *num* between 0 and *num*.

USES

1. Truncating Towards Zero. If *int* is a variable it will be bound to the nearest integer to the number *num* (truncating towards zero). For example, the call

```
int(1.9,X)
```

will bind X to 1. The call

```
int(-134513.456,X)
```

will bind X to the integer -134513.

2. Checking. `int` will succeed if *int* is an integer and it is the nearest integer to the number *num* when *num* is rounded towards zero. For example, the call

```
int(3.2,3)
```

will succeed. The call

```
int(3.6,4)
```

will fail.

6.4 **sign** - the sign of a number

sign(*num*, *sign*)

ARGUMENTS

num	: number
sign	: variable, or -1 or 0 or 1

DECLARATIVE READING

sign = 1 if *num* is greater than 0,
sign = 0 if *num* is equal to 0,
sign = -1 if *num* is less than 0.

USES

1. Finding the Sign of a Number. If *num* is a number (integer or floating point), and *sign* is an unbound variable, the variable will be bound to the sign of the number.

For example, the call

```
sign(-3,S)
```

binds S to -1,

```
sign(0,X)
```

binds X to 0, and

```
sign(25.67,Z)
```

will bind Z to 1.

2. Checking the Sign of a Number. If both *num* and *sign* are given, sign will succeed if *num* has sign *sign*.

For example, the call

```
sign(21.4,1)
```

will succeed. The call

```
sign(21,-1)
```

will fail.

6.5 `ln` - natural logarithm/anti-log

`ln(number, log)`

ARGUMENTS

number	: number or variable (if *log* is not a variable)
log	: number or variable (if *number* is not a variable)

DECLARATIVE READING

$number = e^{log}$

USES

1. To find the natural log of a number. *number* is given and *log* is a variable.

2. To compute a power of e. *log* is given and *number* is a variable.

6.6 `pwr` - power to any base

`pwr(n, p, n_to_p)`

ARGUMENTS

n	: number or variable (if *n_to_p* is not a variable)
p	: number
n_to_p	: number or variable (if *n* is not a variable)

DECLARATIVE READING

$n\_to\_p = n^p$

USES

1. To compute a power. *n* and *p* are given. *n_to_p* is a variable.

For example, the call

```
pwr(2,3,X)
```

will bind X to 8.

2. To find a base. *p* and *n_to_p* are given, *n* is a variable.

For example the call

```
pwr(N,4,81)
```

will bind N to 3.

6.7 `sin` - the sine of an angle in radians

`sin(angle, sine)`

ARGUMENTS

`angle`	: a number or variable (if *`sine`* is a number)
`sine`	: a number in the range -1 .. 1, or a variable (if *`angle`* is a number)

DECLARATIVE READING

`sine` = sine(*`angle`*) where *`angle`* is expressed in radians.

USES

1. Finding the sine. In this case the *`angle`* argument must be given in radians, and the *`sine`* argument will be bound to the value of sine(*`angle`*).

2. Finding the arcsine. If *`angle`* is a variable and *`sine`* is given, *`angle`* will be given the arcsine value in radians in the range - $\pi/2$ to $\pi/2$.

NOTE

2π radians is 360°. The primitive `pi` returns the value of π used by MacPROLOG, and `deg_rad` may be used to convert between degrees and radians - see below.

6.8 `cos` - the cosine of an angle in radians

`cos(angle, cosine)`

ARGUMENTS

`angle`	: a number or a variable (if *`cosine`* is number)
`cosine`	: a number in the range -1 .. 1 or a variable (if *`angle`* is number)

DECLARATIVE READING

`cosine` = cosine(*`angle`*) where *`angle`* is expressed in radians.

USES

1. Finding the cosine. In this case the *`angle`* argument must be given in radians, and the *`cosine`* argument will be bound to the value of cos(*`angle`*).

2. Finding the arcosine. If *`angle`* is a variable and *`cosine`* is given, then *`angle`* will be bound to the value of the arcosine of *`cosine`* in the range - $\pi/2$ to $\pi/2$.

6.9 **tan** - the tangent of an angle in radians

tan(*angle*, *tangent*)

ARGUMENTS

`angle` : a number or variable (if `tangent` is a number)
`tangent` : a number or a variable (if `angle` is a number)

DECLARATIVE READING

`tangent` = tangent(`angle`) where `angle` is expressed in radians.

USE

1. Finding the tangent. The `angle` argument must be given in radians, and the `tangent` argument will become the value of tan(`angle`).

2. Finding the arctangent. If `tangent` is given and `angle` is a variable then the arctangent will be computed and returned as the value of `angle` in the range - $\pi/2$ to $\pi/2$.

6.10 **deg_rad** - conversion between degrees and radians

deg_rad(*d_angle*, *r_angle*)

ARGUMENTS

`d_angle` : number of degrees, or a variable
`r_angle` : number of radians, or a variable

DECLARATIVE READING

`r_angle` = `d_angle` * $2\pi / 360$.

USES

1. Converting to radians. The `d_angle` argument must be given in degrees; `r_angle` will become the equivalent value in radians.

2. Converting to degrees. The `r_angle` argument must be given in radians; `d_angle` will become the equivalent value in degrees.

3. Checking If both arguments are given as numbers, `deg_rad` succeeds if `r_angle` is `d_angle` converted to radians, and fails otherwise.

6.11 **pi** - the value of π

pi(*pi_val*)

ARGUMENTS

pi_val : variable

USE

The variable *pi_val* will be instantiated to the value of π used by MacPROLOG.

6.12 /\ - logical *and* of two integer bit strings

/\(*int1,int2,and_int*) (500 yfx)

ARGUMENTS

int1	: an integer
int2	: an integer
and_int	: a variable

DECLARATIVE READING

and_int is the integer that results from taking the logical *and* of the bit strings of the integers *int1* and *int2*.

6.13 \/ - logical *or* of two integer bit strings

\/(*int1,int2,or_int*) (500 yfx)

ARGUMENTS

int1	: an integer
int2	: an integer
or_int	: a variable

DECLARATIVE READING

or_int is the integer that results from taking the logical *or* of the bit strings of the integers *int1* and *int2*.

6.14 \ - logical complement of an integer bit string

`\(int1,int2)` (500 fx)

ARGUMENTS

int1 : an integer
int2 : a variable

DECLARATIVE READING

int2 is the integer that results from complementing each bit of *int1*.

6.15 » or >> - shift right

`»(integer,shift,result)` (400 yfx)
`>>(integer,shift,result)` (400 yfx)

ARGUMENTS

integer : an integer
shift : a positive integer
result : a variable

DECLARATIVE READING

result is the integer which has the bit pattern representation of *integer* shifted *shift* bit positions to the right.

6.16 « or << - shift left

`«(integer,shift,result)` (400 yfx)
`<<(integer,shift,result)` (400 yfx)

ARGUMENTS

integer : an integer
shift : a positive integer
result : a variable

DECLARATIVE READING

result is the integer which has the bit pattern representation of *integer* shifted *shift* positions to the left.

7 Comparison of Terms

7.1 = - unifiable

`t1 = t2` (700 xfx)

ARGUMENTS

t1, *t2* : any terms

DECLARATIVE READING

t1 and *t2* are equal.

USE

To test that two terms are unifiable. It is defined by:

```
X = X.
```

7.2 \= - not unifiable

`t1 \= t2` (700 xfx)
`t1 ≠ t2` (700 xfx)

ARGUMENTS

t1, *t2* : any terms

DECLARATIVE READING

t1 and *t2* are not equal.

USE

To test that two terms are not unifiable. It is equivalent to

```
not t1 = t2.
```

7.3 == - the identity tester

t1 == t2 (700 xfx)

ARGUMENTS

t1,t2 : any terms

DECLARATIVE READING

t1 and t2 are syntactically identical.

USE

To test that two terms are identical without the need to bind variables in either term. The call will only succeed if the terms have identical structure and contain the same numbers, atoms and variables.

7.4 \== - the non-identity tester

t1 \== t2 (700 xfx)

ARGUMENTS

t1,t2 : any terms

DECLARATIVE READING

t1 and t2 are not syntactically identical.

USE

It is equivalent to the call

```
not t1 == t2.
```

7.5 `unify` - unification with the occurs check

`unify(t1,t2)`

ARGUMENTS

`t1,t2` : any terms

USE

This is the same as = except that the `unify` only succeeds if *t1* and *t2* can be unified without any variable being instantiated to a term containing itself.

For example, both the calls

```
foo(X) = Y
```

and

```
unify(foo(X),Y)
```

will succeed and bind `Y` to `foo(X)`.
However, the call

```
foo(X) = X
```

would result in `X` being bound to the "infinite" term

```
foo(foo(foo( ...
```

but the call

```
unify(foo(X),X)
```

will fail.

7.6 **`compare`** - general term comparison

`compare(`*`rel`*`,`*`term1`*`,`*`term2`*`)`

ARGUMENTS

rel	: variable or one of <, =, >
term1	: any term
term2	: any term

USES

1. If `rel` is given the two terms are compared as specified. = forces a == comparison.

2. If `rel` is a variable the terms are compared and `rel` is bound to `<`, `=` or `>` to indicate the result.

The order relation used is:

variables < atoms < lists < compound terms

Variables are compared by their machine address.

Numbers are compared by value using the < primitive.

The empty list is shorter than any non-empty list. If two non-empty lists have the same first element then their tails are compared.

Compound terms are ordered first by arity, then by the name of their principal functor, and then by each of the arguments in left to right order.

Atoms are compared lexicographically using the international character code convention. This character ordering is somewhat different to the ASCII ordering for characters: when two atoms are compared initially, case is ignored: that is lowercase `a` and uppercase `A` are both regarded as being smaller than lowercase `b` and uppercase `B`.
(In normal ASCII `a < b` and `A < B` but `B > a`).

Where two letters differ only in case then the uppercase letter is 'smaller' than the lowercase letter. So, the following inequalities hold for atoms:

`ab < ac     Ab < ab     ab < Ac`

The < predicate also takes into account certain character decorations. Initially, accented characters such as à õ or é are treated as being the same as the unaccented character, except that when letters differ only by a decoration then the ordering is as follows

a < á < à < â < ä < ã < å

(and similarly for other letters).

7.7 @> - greater than term comparison

t1* @> *t2 (700 xfx)

ARGUMENTS

t1,t2 : any terms

USE

Equivalent to `compare(>,t1,t2)`.

7.8 @< - less than term comparison

t1* @< *t2 (700 xfx)

ARGUMENTS

t1,t2 : any terms

USE

Equivalent to `compare(<,t1,t2)`.

7.9 @>= - greater than or equal term comparison

t1* @>= *t2 (700 xfx)

ARGUMENTS

t1,t2 : any terms

USE

Equivalent to `not compare(<,t1,t2)`.

7.10 @=< - less than or equal term comparison

t1* @=< *t2 (700 xfx)

ARGUMENTS

t1,t2 : any terms

USE

Equivalent to

```
not compare(>,t1,t2).
```

7.11 `mem` - find a subterm corresponding to a path

`mem(`*`term,path,subterm`*`)`

ARGUMENTS

`term`	: any term
`path`	: list of positive integers
`subterm`	: variable

USE

The length of the *`path`* argument specifies the depth of the subterm, and the integers in the *`path`* list specify elements of successive subterms.
For example, the call

```
mem(foo(a,bar(X,b)),[1],M)
```

will bind `M` to `foo`.
The call

```
mem(foo(a,bar(X,b)),[3,1],M)
```

will bind `M` to `bar`.
The call

```
mem(foo(a,bar(X,b)),[3,3],M)
```

will bind `M` to `b`.

The call will fail if the *`path`* refers to a non-existent subterm.
For example, both the calls

```
mem(foo(a,bar(X,b)),[3,8],M)
```

and

```
mem(foo(a,bar(X,b)),[3,1,3],M)
```

fail.

7.12 **`sort`** - sort a list

`sort(`*`list,sortedlist`*`)`
`sort(`*`list,sortedlist,keyselect`*`)`
`sort(`*`list,sortedlist,keyselect,dir`*`)`

ARGUMENTS

`list`	: any complete list of terms
`sortedlist`	: variable, will become sorted version of *`list`*
`keyselect`	: list of positive integers
`dir`	: 1 or `-1`, specifies direction of the sort

USES

1. Two argument call. *`sortedlist`* is bound to a sorted copy of *`list`* with terms in ascending order. The terms are compared using the term comparison primitive @<.

2. Three argument call. *`sortedlist`* is bound to a sorted copy of *`list`* with terms in ascending order.
The terms are compared by comparing a key in each term using the term comparison primitive @<. The key is selected using the *`keyselect`* list of integers. The length of this list indicates the depth of the key. Thus `[1,2]` selects a key at depth 2. The `1` selects the functor (or head element) of the compound terms (or lists) that are the elements of *`list`* and then the `2` selects the first argument (or the second list element) of this term to give the level two key.

Example:

```
sort([g(h(2,5)), k(j(4,3,a),7), [6,m(0,1)]], X, [2,3])
```

binds X to `[[1,m(0,1)],k(j(4,3,a),7), g(h(2,5))]` because 1<3<5.

A value of 1 on the *`keyselect`* list of integers selects the functor of a compound term. So the *`keyselect`* list `[1]` sorts the terms on *`list`* by comparing their functors. *`keyselect`* can also be the empty list.

The call

```
sort(list,sortedlist,[])
```

is equivalent to

```
sort(list,sortedlist).
```

3. Four argument call. Same as the three argument call except that the last argument specifies the direction of the sort. `-1` specifies an ascending order sort (the default if this argument is not given) and `1` specifies a descending order sort.

NOTE Remember that in an operator term the operator will be the *first* element of the term, even if it is *written* as if it were the second element. For example, the term

```
3+5
```

is actually treated as `+(3,5)`.

7.13 **keysort** - sort a list of terms of the form *t-t'* using *t* as a key

`keysort(list, sortedlist)`
`keysort(list, sortedlist, dir)`

ARGUMENTS

`list`	: any complete list of terms of the form `t-t'`
`sortedlist`	: variable, will become sorted version of `list`
`dir`	: 1 or -1, specifies direction of the sort

USES

1. Two argument call. `sortedlist` is bound to a sorted copy of `list`. The terms are compared using the first argument `t` of the operator term `t-t'` as the key.

For example,

```
keysort([3-sam,2-bill,5-andrew],X)
```

binds X to `[2-bill,3-sam,5-andrew]`.

2. Three argument call. Same as two argument call except that the list is in descending order if `dir` has value `1`. A `dir` value of `-1` specifies ascending order.

NOTE

The call

```
keysort([3-sam,2-bill,5-andrew],X)
```

is equivalent to the call

```
sort([3-sam,2-bill,5-andrew],X,[2])
```

This is because the operator terms of the form `t-t'` are treated as being of the form `-(t,t')`.

7.14 append - concatenation relation for lists

append(*list1*,*list2*,*list3*)

ARGUMENTS

list1	: list or variable
list2	: list or variable
list3	: list or variable

DECLARATIVE READING

list3 is *list1* followed by *list2*

DEFINITION

The program append is defined as follows:

```
append([],L,L).
append([E|L1],L2,[E|L3] :- append(L1,L2,L3).
```

For example, the call

```
append([1,2,3], [4,5,6], X)
```

will bind X to the list [1,2,3,4,5,6].

The call

```
append(A, [e|B], [a,p,p,e,n,d]
```

will bind A to the list [a,p,p] and B to the list [n,d].

7.15 on - membership of a list

on(*el*, *list*)

ARGUMENTS

el	: any term
list	: list or variable

DECLARATIVE READING

el is an element on the list *list*

DEFINITION

The program on is defined as follows:

```
on(E, [E|L]).
on(E1, [E2|L])    :- on(E1, L).
```

7.16 length - find the length of a list

length(*list*, *integer*)

ARGUMENTS

list	: a list
integer	: a variable or an integer

DECLARATIVE READING

integer is the length of the list *list*

USES

To find or check the length of a list.
For example, the call

```
length([A, foo(b), c, 1], X)
```

will bind X to 4.

7.17 **remove** - remove an item from a list

remove(*item*, *list*, *remainder*)

ARGUMENTS

item	: any term
list	: variable or list
remainder	: variable or list

DECLARATIVE READING

remainder is the list *list* with the element *item* removed.

EXAMPLES

The call

```
remove(s, [d,u,s,t,y], L)
```

will bind `L` to the list `[d,u,t,y]`.

The call

```
remove(1, L, [2,3,4,5])
```

will bind `L` to the list `[1,2,3,4,5]`.

The call

```
remove(E,[2,3,4],L)
```

will bind `E` to `2` and `L` to `[3,4]`. On backtracking `E` will take the values `3` and `4` with the corresponding values of `L` being `[2,4]` and `[2,3]`.

The call

```
remove(r, [d,u,s,t,y], L)
```

will fail.

7.18 **`reverse`** - reverse a list

`reverse(`*`list`*`,`*`revlist`*`)`

ARGUMENTS

`list`	: variable or list
`revlist`	: variable or list

DECLARATIVE READING

`revlist` is the list *`list`* with the order of its elements reversed.

EXAMPLES

The call

```
reverse([p,a,r,t,s], L)
```

will bind `L` to the list `[s,t,r,a,p]`.

8 Miscellaneous String Operations

In MacPROLOG strings can be represented and manipulated as lists of single character atoms or as lists of bytes. A string in packed form is an atom.

There is a primitive `stringof` that can be used to convert between atoms and lists of characters, in addition to the usual `name` which converts between atoms and lists of bytes.

The primitive `charof` can be used to convert between single character atoms and their ASCII byte codes.

Other string processing primitives are `concat` which joins the print names of two simple terms together to form a new atom, and `gensym` which generates new atoms based on a given root.

Also provided are the primitives `version`, `date` and `time` which give access to the current version of MacPROLOG, and the current date and time expressed either numerically or as strings.

8.1 `stringof` - converts between character lists and atoms

`stringof(`*`chars,atom`*`)`

ARGUMENTS

`chars`	: variable or list of characters or a list pattern
`atom`	: variable or atom

`chars` can only be a variable or a list pattern (which may contain variables) if *`atom`* is an atom. *`atom`* can only be a variable if *`chars`* is a fully instantiated list of characters.

DECLARATIVE READING

`chars` is the list of characters of the atom *`atom`*.

USES

1. Checking. If the call is variable free, `stringof` will succeed if the first argument is the list of characters of the second argument. For example, the call

```
stringof([l,i,s,t],list
```

will succeed.

2. Unpacking. Produces a list of characters from an atom. In this use the second argument must be an atom at the time of evaluation. The first argument is unified with the list of characters of the atom. If the empty atom '' is given its list of characters is the empty list `[]`.

For example,

```
stringof(List,fred)
```

results in `List` being bound to the list `[f,r,e,d]`, and

```
stringof(List2,'A*')
```

binds `List2` to the list `['A','*']`.

For the unpacking use, the first argument may also be a list pattern. This allows particular characters of the atom to be picked up as the binding of variables in the list pattern.

```
stringof([f, r, Ch, d], fred)    Ch = e
stringof([f, r|Chrs], fred)      Chrs = [e, d]
stringof([f, r|Chrs], gerry)     fails
```

3. Packing. This takes a list of characters and produces an atom from it. It is the inverse of the unpack use.

```
stringof([f, r, e, d], Con)        Con = fred
stringof([], Con)                  Con = ''
```

8.2 **charof** - converts between a single character atom and its ASCII code

charof(*char*, *code*)

ARGUMENTS

char : single character atom, or variable (if *code* is not a variable)
code : integer, or variable (if *char* is not a variable).

DECLARATIVE READING

The character atom *char* has the numeric code *code* . The *code* is essentially ASCII except that where extra characters such as Σ ® † é are used the code is specific to the Macintosh (see the Appendices).

USES

1. Checking. If *char* is a single character atom and *code* is an integer, charof will succeed if *code* is the character code for *char*.
For example, the call

```
charof('A',65)
```

will succeed, but the call

```
charof(a,65)
```

will fail.

2. Finding the numeric code of a Character. If *char* is a single character atom and *code* is a variable, the latter will be bound to the code for *char*.
For example, the call

```
charof(a,Code)
```

will cause Code to be bound to 97.

Note that is can also be used for this purpose. The call

```
Code is "a"
```

will also bind Code to 97 because "a" is the list [97] comprising the integer byte code of a and is evaluates a list of a single integer to that integer.

3. Finding the Character Represented by a particular Code. If *char* is a variable, and *code* an integer, *char* will be bound to the character represented by the integer.
For example, the call

```
charof(Ch,98)
```

will bind Ch to the single character atom b.

8.3 name - convert between simple terms and strings represented as lists of ASCII codes.

name(*simple_term*,*list_of_bytes*)

ARGUMENTS

simple_term	: an atom. number or a variable (if *list_of_bytes* is not a variable)
list_of_bytes	: a list of integers which are byte codes, or a variable (if *simple_term* is not a variable)

DECLARATIVE READING

The *simple_term* has an unquoted print name comprising the sequence of ASCII codes on *list_of_bytes*. Remember that a list of bytes is the representation of a string term - a sequence of characters surrounded by double quote marks, such as "apple" (see the chapter on Syntax).

USES

1. To generate or check a list of bytes. If the *simple_term* argument is given, this is converted into a list of byte codes of the characters that make up the (unquoted) print name of the term, and this list of bytes is unified with *list_of_bytes*.
For example, the call:

```
name(tom,String)
```

would result in the variable String being bound to the list [116,111,109] (which is the string "tom").

2. To convert a list of bytes into an atom. If the *list_of_bytes* argument is not a variable but a list of byte codes, and *simple_term* is an unbound variable, then *list_of_bytes* is compressed into an atom which becomes the value of *simple_term*.
For example, the call

```
name(Num,"123")
```

will result in Num being bound to the atom '123'.

8.4 concat - construct a new atom from two simple terms

concat(*sim1*, *sim2*, *atom*)

ARGUMENTS

sim1	: simple term (atom, integer, floating point number or variable)
sim2	: a simple term
atom	: a variable

DECLARATIVE READING

The atom *atom* comprises the print names of the two simple terms *sim1* and *sim2* appended together.

USE

Only one mode of use is supported. Two simple terms can be 'glued' together to form a new atom. Normally the terms to be glued together are atoms: for example the call

```
concat(fred,die,X)
```

would result in the variable X being bound to the atom freddie.

Another common use is to append digits to a basic atom as a form of symbol generator:

```
concat(label,23,Y)
```

results in Y being bound to the new atom label23 (but see gensym below).

PRAGMATIC CONSIDERATIONS

The limit of the size of a single atom is 255 characters. If you try to construct a new atom with more than 255 characters the extra trailing characters will simply be ignored.

8.5 `gensym` - generate the next symbol with a given root
`init_gensym` - initialise the subscript count for a given root

`gensym(`*`root`*`,`*`symbol`*`)`
`init_gensym(`*`root`*`)`

ARGUMENTS

`root` : an atom
`symbol` : a variable (will be bound to an atom)

SIDE EFFECTS

The `gensym` call binds the *`symbol`* variable to the next atom in the sequence

`root0, root1, ...,rootn` .

Each call to `gensym` returns the next atom in the sequence for its *`root`* starting at *`root0`* and at the same time it increments the integer subscript to be used for the next call. A call to `init-gensym` for the same *`root`* will reinitialise the subscript to 0.

The maximum size atom that can be generated is 255 characters.

USE

There is only one supported use for these primitives: to return the next symbol for a given *`root`* with `gensym`, and to reinitialise the sequence of symbol names for a given *`root`* with `init-gensym`.

Example sequence of uses:

`gensym(level,Sym1)`	/* `Sym1` is bound to atom `level0` */
`gensym(level,Sym2)`	/* `Sym2` is bound to atom `level1` */
`init-gensym(level)`	/* reinitialise subscript count for root `level` */
`gensym(level,Sym3)`	/* `Sym3` is bound to atom `level0` */

8.6 pname - find the print name of a term

`pname(term, name)`

ARGUMENTS

term	: any term
name	: a variable or atom

USE

This primitive converts between a Prolog term and the atom representing its print name. For example, the call

```
pname(foo(a,1+2),X)
```

will bind `X` to the atom `'foo(a,1+2)'`.

The calls

```
sqrt(289,Y),
pname(Y,Z).
```

will bind `Z` to the atom `'17'`.

8.7 version - find the current version of MacPROLOG

`version(vers)`

ARGUMENTS

vers	: variable

USE

This primitive returns an atom which gives information on the version of MacPROLOG in use. The call

```
version(X)
```

will bind `X` to an atom similar to the following:

```
'LPA MacPROLOG™ Version 2.0, © 1987 LPA Ltd'
```

8.8 **`date`** - find current date, or check date

`date(`*`year, month, day`*`)`
`date(`*`year, month, day, strdate`*`)`

ARGUMENTS

year	: variable or integer
month	: variable or integer
day	: variable or integer
strdate	: variable

USE

1. Three argument form

If all three arguments are variables, this primitive returns the current date as three integers.
If all three arguments are integers, the call checks that these integers represent a valid date (for example it checks that *month* is an integer between 1 and 12, etc.). Any date may be given - there is no check that it is the current date.

2. Four argument form

The final *strdate* argument must be a variable.
If the first three arguments are all variables, the current date is returned as in the three argument form, and is also returned as an atom in *strdate*.
If the first three arguments are all integers, the date is converted to string form and returned as an atom in *strdate*. Note that 1900 is subtracted from the year on conversion, so that dates before 1900 will yield strange results!

EXAMPLES

If the current date is July 11th, 1987, then the call

```
date(Y,M,D)
```

will bind `Y` to `1987`, `M` to `7` and `D` to `11`.

The call

```
date(1956,3,13,D)
```

will bind `D` to the atom

```
'13th March 56'
```

The call

```
date(1987,4,32)
```

will fail.

8.9 **`time` - find current time, or check time**

```
time(hours,mins,secs)
time(hours,mins,secs,strtime)
```

ARGUMENTS

`hours`	: variable or integer
`mins`	: variable or integer
`secs`	: variable or integer
`strtime`	: variable

USE

1. Three argument form
If all three arguments are variables, this primitive returns the current time as three integers.
If all three arguments are integers, the call checks that these integers represent a valid time (for example it checks that *`mins`* is an integer between 0 and 59, etc.). Any time may be given - there is no check that it is the current time.

2. Four argument form
The final *`strtime`* argument must be a variable.
If the first three arguments are all variables, the current time is returned as in the three argument form, and is also returned as an atom in *`strtime`*.
If the first three arguments are all integers, the time is converted to string form and returned as an atom in *`strtime`*. (The *`secs`* value is not included in this string form.)

EXAMPLES

If the current time is 3:20:11 pm, then the call

```
time(H,M,S)
```

will bind `H` to `15`, `M` to `20` and `S` to `11`.

The call

```
time(18,5,30,T)
```

will bind `T` to the atom

```
'6.05pm'
```

The call

```
time(19,4,62)
```

will fail.

9 Data Base Primitives

The usual Edinburgh interpreted data base primitives `assert` and `clause` have been generalised in MacPROLOG to allow for the storing and retrieval of ground clauses - clauses containing atoms which are specified to be variable names. In fact, all MacPROLOG interpreted clauses are stored as ground clauses with an associated list of variable names. If a hollow clause - a clause containing variables - is added to the data base the variables are given unique underscore names and these are stored as the associated variable names.

There are also additional primitives `assertx` and `clausex` which allow storing and retrieval of either hollow or ground clauses at specified positions in the list of clauses for a relation.

The ability to store ground clauses, combined with the ground read and ground to hollow conversion primitives, is what allows MacPROLOG to remember user variable names in interpreted clauses. It also allows you to build applications in which entered variable names are remembered. You can even build applications which use a quite different variable name convention, for example having as variable names only those atoms that begin with `any` or `some`. All you need is a program to extract the list of the atoms that are the application variable names in a read-in ground term. You can then use `tohollow` to convert it into a hollow term, with just these atoms replaced by variables. Alternatively, if the read-in term represents a fact or rule you can compile it into a clause term and store it with the extracted list of atoms as *its* associated variable names. When the clause is used in an evaluation, or retrieved using `clause`, these names will automatically be replaced by variables.

You can initialise the interpreted data base with the clauses produced by compiling the program text of one or more **Interpreted** or **Data** program windows. The text of a **Data** window will be updated if the definition of any of its relations is changed using a data base primitive. The text of an **Interpreted** window will not be changed when the interpreted data base is changed. You can also initialise the interpreted data base using the `consult` primitive - see below.

The clauses of the interpreted data base are actually compiled. However, each clause for a relation is compiled independently, which is why the relation definition can be updated using `assert` and `retract`. The clauses are also not fully compiled. Information is left in the compiled code so that they can be reverse compiled, enabling `clause` and `listing` to be implemented.

Because the clauses are compiled, *dynamic* is probably a better name for the MacPROLOG clause data base. We use the name *interpreted* because the ability to retrieve the source term using `clause` enables interpreters to be written. It is why full tracing is possible for relations defined in **Data** or **Interpreted** windows.

Code generated for **Compiled** and **Optimised** windows is produced by compiling all the clauses for a relation at the same time. Consequently it is not dynamic: it cannot be updated by adding and deleting individual clauses. Moreover, the code generated cannot be reverse compiled, so the `clause` primitive cannot be used. Code generated from **Compiled** and **Optimised** windows is *fully compiled* code.

You should not try to add clauses for a fully compiled relation. If you do, even if you are adding a clause with a different arity than that of the existing compiled definition, you will get an error message "Adding clause for a compiled relation" (error 14). This is because MacPROLOG does not distinguish between different arities for a relation name. This is also the reason why all the clauses defining a particular relation must be in a single edit window.

The `save` primitive can be used to save all the clauses of the interpreted data base in source form in a file. The `csave` primitive can be used to save all the compiled code - the code for *both* compiled and interpreted programs - in a file.

You can get rid of a fully compiled definition, or *all* the interpreted clauses for a given relation name, using the `kill` primitive.

The primitives `def`, `idef`, `cdef` and `sdef` provide tests for defined relations, and the four dictionary relations `dict`, `idict`, `cdict` and `sdict` allow you to find the names of all the defined relations.

9.1 **assert** or **assertz** - add a new last clause

assert(*clause*)
assert(*gclause*, *varnames*)

ARGUMENTS

clause	: term, a valid clause
gclause	: ground term, a valid clause
varnames	: any list of atoms, specifying the variable names of *gclause*

USES

1. Single argument call. This adds *clause* as a new last clause for the relation name of the head of the clause to the interpreted data base. All the existing clauses remain. The variables in *clause* are given unique underscore names and the clause is added to the data base with these underscore names as the associated list of variable names. *clause* should not contain any (quoted) atoms beginning with underscore.

2. Two argument call. Adds *gclause* as a new last clause for the relation name of the head of the clause to the interpreted data base. *gclause* must be a variable free (ground) clause. The *varnames* list of atoms is stored as the associated list of variable names for the clause.
Usually *gclause* and *varnames* will have been returned by some call of `gread` or `prompt_gread` or a ground read from a dialogue field, but they can also be constructed by some arbitrary MacPROLOG evaluation. The variable names do not need to be Edinburgh syntax variable names.

Example use:

```
assert((likes(keith,any_person):-
        likes(any_person,logic)),[any_person])
```

will add the clause

```
likes(keith,any_person):-likes(any_person,logic)
```

with `any_person` as the only associated variable name.

The call

```
assert(likes(pooh,honey))
```

will add the assertion

```
likes(pooh,honey)
```

with no associated variable names.

`assertz` is an accepted synonym for `assert`. For either use the call fails if the clause argument is not a valid Edinburgh syntax clause.

ERRORS

Invalid form of use error will be signalled for the two argument call if *gclause* is not ground or *varnames* is not a list of atoms. It will also be signalled for the one argument use if *clause* contains any (quoted) atoms beginning with underscore.

9.2 **asserta** - add a clause at beginning of current list of clauses

asserta(*clause*)
asserta(*gclause*, *varnames*)

ARGUMENTS

clause	: term, a valid clause
gclause	: ground term, a valid clause
varnames	: any list of atoms, specifying the variable names of *gclause*

USES

The same two uses as `assert` except that the clause is added at the beginning of the current list of clauses for its head relation.

ERRORS

Same as for `assert`.

9.3 **assertx** - add a clause at a specified position

assertx(*clause*, *index*)
assertx(*gclause*, *varnames*, *index*)

ARGUMENTS

clause	: term. a valid clause
gclause	: ground term, a valid clause
varnames	: any list of atoms, specifying the variable names of *gclause*
index	: integer

USES

The same two uses as `assert` except that the clause is added at the *index*'th position in the current list of clauses for its head relation.

Clauses for a relation are indexed starting at 1, so that

```
assertx(clause,1)
```

is equivalent to

```
asserta(clause)
```

If *index* is greater than the number of clauses for the relation, the new clause is added at the end of the list.

ERRORS

Same as for `assert`.

9.4 `clause` - retrieve a matching clause body

`clause(`*`head`*`,`*`body`*`)`
`clause(`*`ghead`*`, `*`gbody`*`, `*`vars`*`)`

ARGUMENTS

head	: a call term
ghead	: a call term
body	: variable, or a term denoting the body of a clause
gbody	: variable, or a term denoting the body of a clause
vars	: variable, will be bound to a list of atoms, the variable names of the clause

USE

1. Two argument call A search is made for the first hollow copy of an interpreted data base clause for the relation name of *head* with a head and body that unifies with *head* and *body*. Variables in *head* and *body* will be instantiated by the match. The hollow copy of a data base clause is made by replacing each atom of the associated list of variable names by a new unique variable.

An empty body is designated by the atom `true`.

On backtracking, the next matching clause will be found. The call fails when there are no other matching clauses.

2. Three argument call Similar to the two argument call, except that *ghead* and *gbody* are unified with the head and body of a ground copy of the data base clause. In the ground copy the variable names are left as atoms. The *vars* list of atoms is the associated list of variable names for the matching ground clause.

9.5 **`clausex`** - retrieve a matching clause body from a position

`clausex(`*`head, body, from, index`*`)`
`clausex(`*`ghead, gbody, vars, from, index`*`)`

ARGUMENTS

head	: any term which denotes a relation call
ghead	: any term which denotes a relation call
body	: variable, or a term denoting the body of a clause
gbody	: variable, or a term denoting the body of a clause
from	: integer, the clause to search from
index	: integer or variable, the position of the matching clause
vars	: variable

USES

This has the same uses as `clause`, except that the search starts at the *from*'th clause in the list of clauses for the relation name of the *head* term. If *from* is 1 the search starts at the first clause.

If the *index* argument is a variable it will become the actual position of the matching clause retrieved.

If the *index* argument is an integer, this must be the same value as *from*, and only the clause at the *index*'th position will be matched with *ghead* and *gbody*.

In the four argument call, *head* and *body* are matched against hollow copies of the data base clauses.

In the five argument call, *ghead* and *gbody* are matched against a ground copy, and *vars* becomes the list of variable names of the matched clause.

9.6 `retract` - delete a matching clause

`retract(`*`clause`*`)`

ARGUMENTS

`clause` : term, a clause pattern

USE

If there is a matching clause in the data base, that clause is deleted and any variables in *`clause`* are instantiated by the match. On backtracking there is an attempt to find another matching clause. The search always starts at the beginning of the list of clauses for the relation name of *`clause`*. So all clauses asserted between the retract and the redo of the call, even if added using `asserta`, are candidates for deletion on the redo. The call fails when there is no matching clause.

If *`clause`* is not a valid clause pattern the call fails.

For example, if the data base contains the clauses

```
    likes(keith,Someone):-likes(Someone,logic)
and likes(pooh,honey)
```

then the call

```
retract((likes(A,B):-C))
```

will firstly retract the first of the above clauses, and

`A` will be bound to `keith`
`B` will be an unbound variable, for example `_1234`
`C` will be bound to `likes(_1234,logic)`

On backtracking, the second clause will be retracted, and

`A` will be bound to `pooh`
`B` will be bound to `honey`
`C` will be bound to `true`

9.7 `retractx` - delete a clause at a specified position

`retractx(`*`name, index`*`)`

ARGUMENTS

`name` : atom, name of an interpreted data base relation
`index` : integer

USE

The *`index`*'th clause for the relation *`name`* is deleted. If *`index`* is 1, the first clause is deleted. On backtracking there is no attempt to redo the call.

9.8 `retractall` - delete all the matching clauses

`retractall(`*`clause`*`)`

ARGUMENT

`clause`	: term, a clause pattern

USE

All the clauses in the interpreted data base which match *`clause`* are deleted in one operation. Variables in *`clause`* are left uninstantiated by the call. On backtracking there is no attempt to redo the call, even though matching clauses may have been asserted. If *`clause`* is not a valid clause pattern the call fails.

9.9 `abolish` - delete all interpreted data base clauses for a relation that have a given arity

`abolish(`*`name, arity`*`)`

ARGUMENTS

`name`	: atom, name of an interpreted data base relation
`arity`	: non-negative integer

USE

All the interpreted data base clauses for relation *`name`* with *`arity`* head arguments are deleted in one operation. On backtracking there is no attempt to redo the call.

9.10 `kill` - delete one or more complete relation definitions

`kill(`*`rels`*`)`

ARGUMENT

`rels`	: atom, or list of atoms

USE

The complete definition, whether interpreted or compiled, for each relation name in *`rels`* is deleted. Any atom which is not a relation name is ignored. This is the only way to get rid of a compiled definition.

9.11 **save** - save interpreted programs as text

```
save(file)
save(file,vol)
save(file,vol,rels)
```

ARGUMENTS

file	: atom, name of a file
vol	: integer, volume identifier
rels	: list of atoms, names of interpreted relations

USES

1. One and two argument calls. Saves all the clauses of the current state of the interpreted data base in source form in a text file `file`. This is on volume `vol` if this argument is given, otherwise it is on the current default volume as recorded by `dvol`. Any existing file on the volume of the same name is overwritten.

2. Three argument call. Saves all the clauses for the given list of intepreted relations in the file `file` on volume `vol`.

9.12 **csave** - save compiled code

```
csave(file)
csave(file,vol)
csave(file,vol,rels)
```

ARGUMENTS

file	: atom, name of a file
vol	: integer, volume identifier
rels	: list of atoms, names of relations

USES

1. One and two argument calls. Saves all the compiled code of all the currently defined user relations (i.e. all the relations returned by `dict`) in a code file `file`. This is on volume `vol` if this argument is given, otherwise it is on the current default volume as recorded by `dvol`. Any existing file on the volume of the same name is overwritten. The saved definitions can be reloaded using the `consult` primitive or the **Load...** command of the programming environment.

Note that the compiled code of all the clauses in the interpreted data base is also saved by this call.

2. Three argument call. Saves all the compiled code for the definitions of the relations in the list `rels` in file `file` on volume `vol`. The `rels` list can be a mixture of compiled and interpreted relations.

9.13 `consult` - load a program from a file

```
consult(file)
consult(file, vol)
```

ARGUMENTS

file	: atom, name of a file
vol	: integer, volume identifier

USE

Load a program from a file. The file can be:

(1) a compiled code file - the code is loaded, overwriting any existing compiled or interpreted definitions for relations with the same names as in the loaded file. So for a compiled code file this is actually a reconsult. Previously saved code for interpreted relations in the *file* is put into the interpreted data base.

(2) a text file of clauses - no edit windows are created but all the clauses in the file are added to the interpreted data base, using `assert`, as they are loaded. They are therefore added to the clauses for existing interpreted data base relations of the same name. Variable names in the read-in clauses are remembered.

ERRORS

When loading a text file the error 'Cannot add clause for compiled relation' (error 14) is signalled if a clause is encountered for a relation which has a compiled definition. If you choose the **Succeed** response, the load will continue but the clause will be ignored. Any other response will cause the load to terminate.

9.14 `xrefs` - find the list of relation names used in a program

```
xrefs(rel_name, list_of_relations)
```

ARGUMENT

rel_name	: atom, name of a relation
list_of_relations	: variable

USE

The variable *list_of_relations* will be bound to a list of all the names of the relations that appear in the program for *rel_name*. A relation name will be included on the list even if it appears in an argument term of a call in the program.

This can be used for compiled programs as well as interpreted programs.

9.15 **`def`** - test if a relation is defined

`def(`*`call`*`)`

ARGUMENT

`call` : call term

USE

To test if an atom is the name of a relation defined by either a compiled or interpreted program, or to test if a compound term has a functor which is a defined relation name.

9.16 **`idef`** - test if a relation is interpretively defined

`idef(`*`call`*`)`

ARGUMENT

`call` : call term

USE

To test if an atom is the name of a relation defined by an interpreted program, or to test if a compound term has a functor which is a interpreted relation name.

9.17 **`cdef`** - test if a relation is defined by a fully compiled program

`cdef(`*`call`*`)`

ARGUMENT

`call` : call term

USE

To test if an atom is the name of a relation defined by a fully compiled program, or to test if a compound term has a functor which is the name of a fully compiled relation.

9.18 **`sdef`** - check if a relation is defined by a system program

`sdef(`*`call`*`)`

ARGUMENT

`call` : call term

USE

To test if an atom is a protected relation name defined by a system program (its name is in the system dictionary), or to test if a compound term has a functor which is in the system defined relations.

9.19 `dict` - find the list of all the user defined relations

`dict(`*`list`*`)`

ARGUMENT

`list` : variable, will be bound to a list of atoms

USE

A list of names of all the relations defined, either by clauses in the interpreted data base or by fully compiled code, is returned as the binding for *`list`*. `dict` returns the list of all the names for which `def` is true.

9.20 `idict` - find the list of all the user defined interpreted relations

`idict(`*`list`*`)`

ARGUMENT

`list` : variable, will be bound to a list of atoms

USE

A list of names of all the relations defined in the interpreted data base is returned as the binding for *`list`*. `idict` returns the list of all the names for which `idef` is true.

9.21 `cdict` - find the list of all the user defined compiled relations

`cdict(`*`list`*`)`

ARGUMENT

`list` : variable, will be bound to a list of atoms

USE

A list of names of all the fully compiled relations is returned as the binding for *`list`*. `cdict` returns the list of all the names for which `cdef` is true.

9.22 `sdict` - find the list of reserved system relation names

`sdict(`*`list`*`)`

ARGUMENT

`list` : variable, will be bound to a list of atoms

USE

Returns the list of all the protected relation names. You are not allowed to enter a definition for any relation name on the `sdict` list.

9.23 `listing` - list a relation

```
listing
listing(pred)
```

ARGUMENTS

pred : atom or list of relation names

USE

In the no argument call, all interpreted relations currently defined are listed to the current output channel.
If the *pred* argument is given, the specified relation or list of relations are listed to the current output channel.

10 Property Management

In addition to the interpreted data base which can be manipulated using the database primitives. MacPROLOG has a property management system similar to that of LISP. With any atom, we can associate any number of properties each of which has a value which can be any term. Any atom or number can be used as a property name. Properties can have their values set, changed and retrieved. MacPROLOG's programming environment makes heavy use of properties. For example, the identity of the edit window in which a relation `rel` is defined is recorded as the value of the property `'DEFINWIN'` of the name `rel`.

In addition to the property manipulation primitives there are two very useful property search primitives. One, `get_cons`, enables you to find all the atoms which have an associated property, the other, `get_props`, enables you to get all the names of properties associated with a given atom.

The advantages of using properties instead of facts in the interpreted data base are compactness of the representation of the *atom property_name value* triples, increased speed of retrieval of the property values, and the fact that you can find the names of all the properties of an atom. The disadvantage is lack of symmetry between the atom name and the property value. If you record the value **V** of a property **P** for an atom **A** as a clause for the relation **P**, then you can retrieve **A** given **V**, or you can retrieve **V** given **A**. If you record it as the value of property **P** for the atom **A**, then you cannot retrieve **A** given **V**, you can only retrieve **V** given **A**.

Finally, the four primitives `remember`, `recall`, `default` and `forget` provide the features of a symbolically named memory location.

Reserved property names

In the Appendices is a list of the property names used by MacPROLOG: you should not use any of them as a property name. If you use them for your own properties you may affect the behaviour of a MacPROLOG primitive or a menu command of the programming environment. The reserved names all use capital letters, so you will avoid a clash if you use only lower case letters as your property names. A property name can be any atom.

10.1 **set_prop** - set a property value

set_prop(*object,property,value*)

ARGUMENTS

object	: atom
property	: atom or integer
value	: any term

SIDE-EFFECT

value becomes the remembered value of the property *property* of the object *object*. Any previously remembered value of the *property* of *object* is lost.

10.2 **get_prop** - retrieve the value of a property

get_prop(*object,property,value*)

ARGUMENTS

object	: atom
property	: atom or integer
value	: any term

USE

The current value of the property *property* of object *object* is retrieved and unified with *value*.
If there is no remembered value, the call fails.

10.3 **del_prop** - remove a property value

del_prop(*object,property*)

ARGUMENTS

object	: atom
property	: atom or integer

SIDE EFFECT

The remembered *property* value of *object* is forgotten. A subsequent call to

```
get_prop(object,property,value)
```

will fail.

`del_prop` does nothing and succeeds if there is no such *object* or *property*.

10.4 `remember` - store a value

`remember(atom, value)`

ARGUMENTS

atom	: an atom
value	: any term

USE

Stores `value` as the currently stored value of `atom`. Any previously assigned value is lost.

10.5 `recall` - retrieve a stored value

`recall(atom, value)`

ARGUMENTS

atom	: an atom
value	: variable or term

USE

The currently stored value of `atom` is retrieved and unified with `value`.
`recall` fails if `atom` has no currently stored value.

10.6 `default` - retrieve a value or a default

`default(atom, value, default)`

ARGUMENTS

atom	: an atom
value	: any term
default	: any term

USE

`value` will be unified with the currently stored value of `atom` if there is one. It will be unified with `default` if there is no currently stored value.

10.7 `forget` - delete a stored value

`forget(atom)`

ARGUMENT

atom	: an atom

USE

To delete the stored value associated with *atom*. A `recall` on *atom* will now fail. A call to `forget` always succeeds, even if there is no stored value.

10.8 `get_cons` - find all the objects with a given property

`get_cons(property,list)`

ARGUMENTS

property	: atom or integer, name of a property
list	: unbound variable

USE

The variable *list* will be bound to the list of all the atoms that currently have a value for the property *property*.

10.9 `get_props` - find all the properties of a given object

`get_props(object,list)`

ARGUMENTS

object	: atom
list	: unbound variable

USE

The variable *list* will be bound to the list of all the names of the properties currently associated with *object*.

11 File and Window I/O

The usual Edinburgh term and byte I/O primitives are available for use in MacPROLOG programs as well as an extra term input primitive `gread`. They can be used for file I/O, window I/O and serial I/O. Windows may be of type `prog`, `disp` or `info` (see the chapter on Window Handling).

As usual, the input / output channel can be set with a call to `see` or `tell`. However, as an alternative, MacPROLOG allows the name of the file or window to be used for a particular I/O operation to be specified in the I/O call. This allows an application to read and write to several windows or files without the need to repeatedly switch channels. If an I/O operation is to be performed on a file using the explicit file name feature, the file must first be opened using an `open` call. If the channel is `printer` or `modem`, it should first be opened and configured by calls to `seropen` and `serconfig` (see the chapter on Serial I/O).

You can write to the **Default Output Window** by giving the reserved window name `user`. In fact, `user` is the default channel to which output is sent if no channel is given as an argument to the output call and no other default output channel has been specified by a `tell` call.

To read from the keyboard, you should use the `prompt_read`, `prompt_gread` or `ask` primitives described in the chapter on Predefined Dialogues, which allow a prompt to be displayed. However, if you use `read` or `gread` with `user` as the current input channel, a similar standard input dialogue will be displayed with the default prompt: `"Enter any term:"`.

11.1 see - set the input channel

`see(`*`channel`*`)`

ARGUMENTS

channel : file or window name

USE

The default input stream is set to *channel*.

If an input channel is not set, `user` is assumed.

With `user` as the input channel only the single argument calls to `read` and `gread` will succeed. Single argument calls to `get`, `get0` and `skip` will all immediately fail.

If *channel* is a file that is not already open, an attempt will be made to open a file of the given name on the current default volume (see `dvol`). If the file is already open, calls to `see` will not reopen it or reposition the file pointer.

11.2 `tell` - sets the output channel

`tell(`*`channel`*`)`
`tell(`*`channel, volume`*`)`

ARGUMENTS

`channel` : file or window name
`volume` : integer, volume identifier

USE

The default output stream is set to *`channel`*.
If *`channel`* is a file name, an optional *`volume`* id can be given. If it is not, the current default volume (see `dvol`) will be assumed.
If the *`channel`* is a file which is not already open, a file of the given name will be created. If the file is already open, the call will not recreate the file or reposition the file pointer.

If an output channel is not set, `user` is assumed.
With `user` as the output channel, output will be sent to the **Default Output Window**.

11.3 `seeing` - finds the input channel

`seeing(`*`channel`*`)`

ARGUMENTS

`channel` : variable

USE

`channel` is bound to the file or window name of the current default input channel.

11.4 `telling` - finds the output channel

`telling(`*`channel`*`)`

ARGUMENTS

`channel` : variable

USE

`channel` is bound to the file or window name of the current default output channel.

11.5 **`seen` - cancel default input channel selection**

`seen`

USE

This cancels the effect of the last `see` call - `user` becomes the current input channel.

If the last input channel was a file, the `seen` primitive will automatically close the file.

11.6 **`told` - cancel default output channel selection**

`told`

USE

This call cancels the effect of the last `tell` call - `user` becomes the current default output channel.

If the last output channel was a file, the `told` primitive will automatically close the file.

11.7 `read` - read a hollow term from an input channel

`read(`*`term`*`)`
`read(`*`channel`*`,`*`term`*`)`

ARGUMENTS

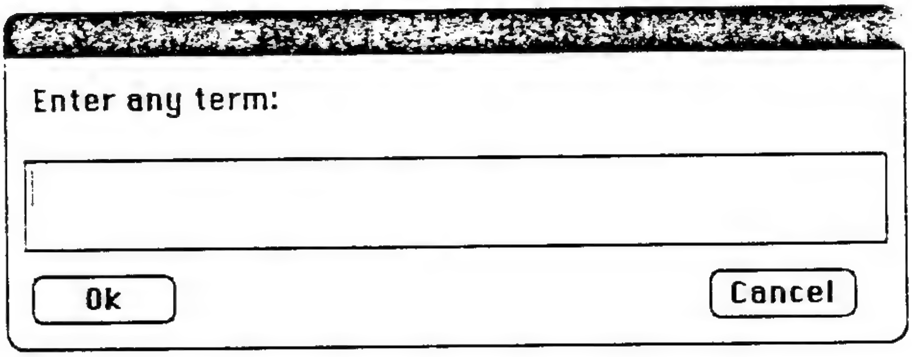
`term`	: variable or term
`channel`	: atom, name of channel

USES

If *`term`* is a variable, it will be bound to the next term on *`channel`*. The term must be terminated by a full stop followed by space or a carriage return, which will be consumed by the call. Any variable names in the read-in term are converted into variables.

If the *`term`* argument is not a variable, an attempt is made to unify the read-in term with *`term`*. The call fails if the unification fails.

For the single argument call, if `user` is the current default input channel, the following modeless dialogue will be displayed



A term should be entered in the edit field of the dialogue. No terminating full stop is required in this case.

When the end of the file or the end of the window is first reached, *`term`* is unified with the atom `end_of_file`. Subsequent calls to `read` will produce the error "Read past end of file or window". If this is signalled you should select the **Stop** or **Fail** response to the error handler dialogue.

Note that a `read` takes place at the current position in *`channel`*. For a window, this is the cursor position, and for a file it is the file pointer. The position is updated after the `read`. See the `cursor` primitive in the chapter on Window Handling and the `seek` primitive in the chapter on File Handling for more information on positioning these file and window pointers.

11.8 gread - read a ground term from an input channel

```
gread(term)
gread(channel, term)
gread(channel, term, varnames)
```

ARGUMENTS

term	: variable
varnames	: variable
channel	: atom, name of channel

USES

This call is similar to `read`, but the difference between `gread` and `read` is that variable names in the read-in term will not be converted into variables. They will appear in *term* as (quoted) atoms.

Where the *varnames* argument is given, this will be bound to the list of atoms that are the variable names of the read-in term.

PRAGMATICS

See the description of `prompt_gread` for an illustration of the usefulness of such a ground read primitive. In can be used in conjunction with the two argument form of `assert` to remember variables of read-in clauses stored in the interpreted data base.

11.9 edintok - read a single token from an input channel

```
edintok(token, type)
edintok(channel, token, type)
```

ARGUMENTS

token	: variable
type	: variable
channel	: atom, name of channel

USE

A single *token* is read from the current input channel, or from *channel* if this argument is given.
The *type* of the token is an integer, indicating one of the following.

0	Separator (space or dot space)
1	Punctuation character
2	Single character symbol (e.g. ! or /)
3	Symbolic name (e.g. £££)
4	Variable (alphanumeric, starting with uppercase letter or underscore)
5	Alphanumeric atom
6	Number
7	String
8	Quoted atom

11.10 `write` - write an operator term to an output channel

```
write(term)
write(channel, term)
```

ARGUMENTS

term	: any term
channel	: atom, name of channel

USE

If the *channel* argument is not given, *term* is written out to the current default output channel, otherwise it is sent to *channel*.

Variables are displayed as underscore names.

Atoms that would need to be quoted on input are *not* quoted.

Current operator declarations are used.

11.11 `display` - write a term to output channel ignoring operators

```
display(term)
display(channel, term)
```

ARGUMENTS

term	: any term
channel	: atom, name of channel

USE

If *channel* is not given, *term* is written out to the current default output channel, otherwise it is sent to *channel*.

Variables are displayed as underscore names.

Atoms that would need to be quoted on input are quoted.

Operator declarations are ignored.

PRAGMATICS

Use this primitive when saving terms to a text file which you will not want to edit. When you read the terms back, input will be faster if you used `display` rather than `writeq` to write the terms, because of the absence of operators.

11.12 `writeq` - write a quoted term to an output channel

```
writeq(term)
writeq(channel, term)
writeq(channel, term, varnames)
```

ARGUMENTS

term	: any term
channel	: atom, name of file or window
varnames	: list of atoms

USE

If *channel* is not given, *term* is written out to the current default output channel, otherwise it is sent to *channel*.

Variables are displayed as _*hex* where *hex* is a hexadecimal number.

Atoms that would be quoted on input are quoted with the exception that, in the three argument call, none of the variable names in *varnames* will be quoted.

Operator declarations are used.

Use the three argument form to display ground data base clauses that have been retrieved using the three argument form of `clause`.

11.13 `nl` - write a carriage return to an output channel

```
nl
nl(channel)
```

ARGUMENT

channel	: atom, name of channel

USE

For the no argument call, a carriage return character (ASCII 13) is sent to the current default output channel. If *channel* is given, it is sent there.

11.14 get0 - get next character from input channel

```
get0(byte)
get0(channel,byte)
```

ARGUMENTS

byte	: variable or integer
channel	: atom, name of channel

USE

If *byte* is a variable, it will be bound to the ASCII code of the next character on the current default input channel if no channel is given, or to the code of the next character on *channel*. The file pointer or window cursor will be moved forward one character.

If the *byte* argument is an integer, the read-in byte is unified with the given *byte*. The call fails if the unification fails.

The call fails if the input channel is `user`.

11.15 get - get next printable character from an input channel

```
get(byte)
get(channel,byte)
```

ARGUMENTS

byte	: variable or integer
channel	: atom, name of channel

USE

In the one argument call, *byte* will be unified with the ASCII code of the next printable character on the current default input channel. In the two argument call, it is the code of the next printable character in *channel*.

The file pointer or window cursor will be moved forward. Non-printable characters with ASCII code ≤ 32 are skipped.

The call fails if the input channel is `user`.

11.16 `skip` - skip to a character in a channel

```
skip(byte)
skip(channel,byte)
```

ARGUMENTS

byte	: integer ASCII code
channel	: atom, name of channel

USE

The file pointer or window cursor will be moved forward to immediately after the next character with ASCII code *byte* in the current default input channel if *channel* is not given, otherwise the forward skip is in *channel*.
The *byte* argument can be an arithmetic expression which evaluates to an integer in the ASCII range. This means that it can be also string of the form `"c"` where `c` is the character to be found. For example,

```
skip(".")
```

will cause a skip to immediately after the next occurrence of the full stop character. (The string `"."` is the list `[46]` comprising the code of a full stop, and as an arithmetic expression the unit list `[46]` evaluates to 46.)
The call fails if the input channel is `user`.

11.17 `put` - write a character to an output channel

```
put(byte)
put(channel,byte)
```

ARGUMENTS

byte	: integer ASCII code
channel	: atom, name of channel

USE

Puts character with ASCII code *byte* to *channel*, or to the current default output channel if the *channel* argument is not given. As with `skip`, *byte* can be an integer expression.

11.18 `tab` - write spaces to an output channel

```
tab(int)
tab(channel,int)
```

ARGUMENTS

int	: integer
channel	: atom, name of channel

USE

int spaces are written to *channel*, or to the current default output channel if this argument is not given.

11.19 `tload` - load a text file into a display window

```
tload(file)
tload(file, volume)
```

ARGUMENTS

file	: atom, name of a text file
volume	: integer, volume ID

USE

An invisible display window with the name *file* is created, into which the text of *file* is then loaded. (This is used by the `help` primitive).

If the *volume* argument is not given the file is searched for in the default volume.

If you subsequently want to make the display window visible and bring it to the front you can make calls to `wshow` and `wfront` after calling `tload`. The window is for display only and cannot be edited.

For example, the calls

```
tload('8 Queens Help'),
wshow('8 Queens Help'),
wfront('8 Queens Help').
```

will load the text file `'8 Queens Help'` into a display window of the same name, and will then make this window visible and bring it to the front.

12 File Handling

In this chapter we describe the facilities available within MacPROLOG for using the Macintosh file system. We concentrate on those primitives needed to access the file system; the primitives which perform the actual input/output are described in the chapter on File and Window I/O.

There are a number of special primitives which invoke standard Macintosh dialogues to select the file to read or write. These allow the programmer to construct MacPROLOG applications which conform to the Macintosh user guidelines.

12.1 `old` - allow the user to select a file name and volume

`old(`*`type, file, volume`*`)`

ARGUMENTS

type	: a file type. It is an atom of up to 4 letters.
file	: variable. Will be the selected file name
volume	: variable. Will be the selected volume id.

USE

A standard file dialogue is displayed on the screen. All the files that are in the current folder with the required type will be displayed. The user can change disks and open and close folders via options in the dialogue.
When the user has selected a file to open, the `old` primitive returns with the name of the file and the integer identifier of the file's volume.

The returned *file* and *volume* values can then be used by the programmer to actually open the file (see below).

The `old` primitive fails if the user clicks on the **Cancel** button on the file dialogue.

PRAGMATICS

The primary use for the `old` primitive is for a program to establish which files the user wishes to process. By specifying the required *type* the program ensures that only those files which are relevant to the program are ever displayed on the screen: this reduces the clutter that the user has to go through. The file types used by MacPROLOG are:

- `'TEXT'` : standard text file
- `'SIGM'` : MacPROLOG compiled code file
- `'PROL'` : MacPROLOG source windows file
- `'PRBT'` : MacPROLOG BOOT file

To display all file types use the empty string '' for the *type* argument.

NOTE It is highly recommended that `old` be used for selecting file names and volumes. The main benefit of using the `old` primitive is that users are never asked to actually type in the name of an existing file; they simply select the file they want from those which are available. It is also the only way to find the volume identifier which is needed in the `open` command if the file to be opened is not in the current default volume.

12.2 `new` - allow the user to create a new file name and select a volume.

`new(file,vol,prompt)`

ARGUMENTS

file	: variable. Becomes an atom which is the name of the file.
vol	: variable. Becomes the volume id on which to create the file.
prompt	: atom. A message saying what sort of file is to be made.

USE

A standard file dialogue is displayed in which the user is invited to enter a new file name in an edit field. The prompt string is displayed to inform the user what sort of file is to be created.

The user has the opportunity to format disks or change disks and folders as well as to enter the file name. If the file already exists then the user is asked to confirm that the old file is to be replaced.

The edit field of the dialogue is preloaded with the name of the previous file that was selected using an earlier call to `old` or `new`. This allows an application to use `old` to offer the user a menu of files which may be consulted, and then to use `new` to allow the user to give a file name into which possibly changed data is to be saved. The default is the name of the file that was last consulted.

As with the `old` primitive the user has the option to **Cancel** the dialogue, in which case the `new` call will fail.

The *prompt* atom will be displayed without quotes.

NOTE
The file selected is not actually `create`-d by the `new` primitive; it is simply a user friendly way to select a file name.
This is the recommended method for allowing the user to specify a file name and volume.

12.3 open - open an existing file for read/write access.

```
open(file)
open(file,vol)
```

ARGUMENTS

`file`	: atom. The name of the file to open.
`vol`	: integer. The identifier of the volume.

USES

The named file is opened. If the file has read/write permission associated with it at the operating system level, it will be opened for read/write access. If the file is write protected, it will be opened for read access only.

If the file was already open the file pointer is repositioned to the beginning of the file.

The `vol` argument will be one that has been returned from a call to `old`.
If the optional `vol` identifier is not given the default volume is assumed, as recorded by `dvol`.

The `open` primitive is used to open a file that already exists. Information can be read from, or written to the file. `open` is used to signal to the MacPROLOG system that you want to start reading or writing to the file.

Note that `open` does not allow you to specify that a file be opened for read-only access. A file will only be opened for read-only access if it is write protected at the operating system level.

No backup copy of the `open`-ed file is made.

The normal use of `open` should be in conjunction with `old`. `old` allows the user to select the file and volume which are then passed as arguments to `open`.

WARNING
Although you can have many files open simultaneously in MacPROLOG you cannot have two files with the same file name (in different volumes) open at the same time. If you try and do this, MacPROLOG will just read and write to the most recently opened file. You will not even be able to close the previously opened file with the same name.
This is because once a file has been opened MacPROLOG identifies the opened file for subsequent I/O operations using just its file name, ignoring the volume.

ERRORS

No such volume (-35):
Disk I/O error (-36):
File not found (-43):
File is already open (-47):
Volume not on line (ejected) (-53):
No such drive (-56):
Directory corrupted (-60):

12.4 **`create`** - create a new version of a file.

```
create(file)
create(file,vol,type)
```

ARGUMENTS

`file`	: atom. The name of the file to create.
`vol`	: integer. The volume id of where to create the file.
`type`	: atom, up to four letters. The file's type.

USES

A new *`file`* of the specified *`type`* is created on the volume *`vol`*. If the file already exists it is truncated to a zero-length file. Subsequent file output calls will extend the file.

If the one argument form of `create` is used the default drive is assumed, as recorded by `dvol`, and the type of the file created is `'TEXT'`. (See the `old` primitive for the allowed file types.)

The normal use of `create` is in conjunction with `new`. The preceding call to `new` is used to determine the *`file`* name and *`vol`* id to use in the `create` call.

ERRORS

Directory is full (-33):
Disk is full (-34):
No such volume (-35):
Disk i/o error (-36):
Too many files (-42):
File already open to write to (-49):
Volume not on line (-53):
File permission error (-54):
No such drive (-56):
Directory corrupted (-60):

12.5 `close` - close a file after processing.

`close(file)`

ARGUMENTS

`file` : atom. The name of the file to close.

USE

The `close` primitive should be used when you have finished reading and writing to a previously `open`-ed or `create`-d file. It is only when the file is actually closed that you can be sure that the disk has been correctly updated with the contents of the file.

If *`file`* is not the name of a currently open file then `close` does nothing and the call succeeds.

ERRORS

Disk is full (-34):
Disk I/O error (-36):
File not open (-38):
Disk is write protected (-44):

12.6 `dvol` - read / set the default volume

`dvol(vol)`

ARGUMENT

`vol` : variable or integer, the volume id.

DECLARATIVE READING

`vol` is the identifier of the current default volume, which can be a disk or a folder.

USES

If the *`vol`* argument is a variable, `dvol` will return the integer id of the current default volume. On entry to MacPROLOG, the default volume will be the disk or folder on which the LPA MacPROLOG™ file resides.

If the *`vol`* argument is an integer, this call will change the current default volume to the specified volume.

Be careful when using this call that you use a valid volume identifier. In general the *`vol`* argument should only be one that has been returned by a previous call to `old` or `new`. If *`vol`* is not a recognised volume identifier, an error will be generated.

ERRORS

No such volume (-35):

12.7 `seek` - report/reposition the file pointer

`seek(`*`file,position`*`)`
`seek(`*`file,mode,position`*`)`

ARGUMENTS

`file`	: atom. Name of a currently open file.
`position`	: integer or variable. Position of the file pointer.
`mode`	: integer - value 1, 2 or 3. Reposition mode (see below)

DECLARATIVE READING

`seek(`*`file,position`*`)`	: The character pointer for *`file`* is at *`position`*.
`seek(`*`file`*`,1,`*`position`*`)`	: The pointer for *`file`* is *`position`* characters from the beginning of the file.
`seek(`*`file`*`,2,`*`position`*`)`	: The pointer for *`file`* is *`position`* characters from the end of the file.
`seek(`*`file`*`,3,`*`position`*`)`	: The pointer for *`file`* is *`position`* characters from the last position of the pointer.

USES

1. Reporting the current position of the file pointer. This is the two argument call in which the *`file`* name is given and *`position`* is a variable. The primitive returns with the current value of the file pointer for that file. *`position`* will be 0 if the pointer is at the start of the file.

2. Changing the position of the file pointer. *`file`* gives the name of the file to reposition, and the *`mode`* indicates in what way the file pointer is to be repositioned. The *`mode`* indicates whether the new position is relative to the start of the file (*`mode`* = `1`), relative to the end of the file (*`mode`* = `2`) or relative to the current file pointer (*`mode`* = `3`).
A call

```
seek(file,pos)
```

where *`pos`* is given is equivalent to the call

```
seek(file,1,pos).
```

The `seek` primitive can be used for files that have been opened using `open` or `create`. Extreme care should be exercised if writing into the middle of a file, as there is no protection against overwriting already existing terms in the file.

`seek` can be used to implement a system where one or more programs can be on disk instead of in memory.

The following is a simple program which allows clauses to be stored in a file; it is the basis of a simple relational data base system.

```
rpred(F,Oldpos,Goal):-          /* find in the file F a clause that matches
                                   the goal Goal, starting the search at Oldpos */
   seek(F,0,Oldpos),            /* reposition file pointer to Oldpos */
   read(F,Cl),                  /* read in a candidate clause Cl */
   seek(F,Newpos),              /* find new file position Newpos */
   (Goal=Cl; Goal:-Bdy =Cl,Body;
   rpred(F,Newpos,Goal)).       /* either unify candidate clause Cl
                                   with Goal returning Body and
                                   evaluate Body or find another clause
                                   starting at Newpos */
```

You can use this program to keep the clauses for a relation `likes` (say) in the disk file "The likes file". To do this replace the clauses for `likes` in the program with the single clause:

```
likes(X,Y) :- rpred('The likes file',0,likes(X,Y)).
```

The file "The likes file" must also have been `open`-ed prior to using the program for `likes`. The "The likes file" file should be a `'TEXT'` file generated by a program that uses `write` to write out the individual clauses onto the file.

The `likes` clauses in "The likes file" can be quite general, including recursive clauses, because different invocations of `rpred` can be accessing different segments of the file simultaneously. The use of `seek` to record the new position of the file after the next clause has been read means that each different use of `rpred` remembers where it is in the file, and then explicitly repositions the file to that position when it is getting its next clause.

You can spread the clauses for a relation over several files (just use more than one rule of the above form) and you can mix `rpred` definitions of relations with ordinary clauses. So, new information about a data base relation can be recorded by a clause in memory until the backing store file can be updated. Programs like `rpred` can be used to build intelligent data base systems within MacPROLOG.

ERRORS

File not open (-38):
Tried to position before start of file (-40):
Seek error (-52):

12.8 **`delete`** - delete a file from the disk.

`delete(`*`file`*`, `*`vol`*`)`

ARGUMENTS

`file`	: atom. The name of the file to delete
`vol`	: integer. The volume id where the file resides.

USES

This primitive deletes the named file from the identified volume. If the file did not exist then the primitive fails.

Use this call after a call to `old` to establish the file name and volume identifier.

ERRORS

No such volume (-35):
Disk I/O error (-36):
Disk is write protected (-44):
File is locked (-45):
Volume is locked (-46):

12.9 **`rename`** - rename a file.

`rename(`*`old_file`*`, `*`vol`*`, `*`new_file`*`)`

ARGUMENTS

`old_file`	: atom. Old name of the file.
`new_file`	: atom. New name of the file.
`vol`	: Integer. Volume where the file resides.

USE

The name of *`old_file`* is changed to *`new_file`*. The *`vol`* parameter gives the identifier of the volume where the file resides.

Use the `old` and `new` primitives to establish what file the user wants to rename, and to what new name.

ERRORS

Renamed file already exists (-48):

12.10 `ftype` - get the type and creator of a file.

`ftype(`*`file, vol, type, creator`*`)`

ARGUMENTS

file	: atom, name of the file.
vol	: integer, volume where the file resides.
type	: variable or atom, the file type.
creator	: variable or atom, the file's creator.

USE

The call may be used to return or check a file's type and its application.

The *type* is an atom of up to four characters (see the description of the `old` primitive for the file types used by MacPROLOG).

The *creator* is an atom of up to four characters and indicates which application created the file. Files created by MacPROLOG have `'SIGM'` as their *creator*.

13 Execution

13.1 `abort` - abort the current process

`abort`

USE

This call cancels the evaluation currently being executed.

13.2 `halt` - exit from MacPROLOG

`halt`

USE

This call will cause an irreversible exit from MacPROLOG.

WARNING If `halt` is called no files will be saved and the `'<QUIT>'` program will not be executed.

13.3 op - declare an operator

`op(`*`priority`*`, `*`type`*`, `*`op_name`*`)`

ARGUMENTS

priority	: integer
type	: atom, operator type
op_name	: atom, operator name

USE

This call declares an operator with name *op_name*.

The *priority* argument should be an integer between 1 and 1200, which will determine the operator precedence when the operator's arguments are themselves operator expressions. The lower the priority number, the more binding the operator (see the Syntax chapter).

The *type* argument is one of:

`fx`	for non-associative prefix operator
`fy`	for right associative prefix operator
`xf`	for non-associative postfix operator
`yf`	for left associative postfix operator
`xfx`	for non-associative infix operator
`xfy`	for right associative infix operator
`yfx`	for left associative infix operator

When several operators have the same declaration, the *op_name* argument can be a list of operator names.

(See the Appendices for the predeclared operators of MacPROLOG.)

13.4 `current_op` - operator test

`current_op(priority, type, op_name)`

ARGUMENTS

priority	: integer or variable
type	: atom or variable, operator type
op_name	: atom or variable, operator name

USE

If *op_name* is an atom, the call can be used to find the *priority* and *type* of a given operator. (See the op primitive for a list of operator types.)

If *op_name* is a variable then the call `current_op` may be used to backtrack through operators, and in this case if either *type* or *priority* is an atom then only operators of that *type* and/or *priority* will be retrieved.

13.5 `spy` - set spypoints

`spy(pred)`

ARGUMENT

pred	: atom or list of atoms

USE

The *pred* argument may be either a single relation name or a list of relation names. The call sets spypoints on the specified relations.
This is equivalent to the **Spypoints** menu command of the programming environment.

13.6 `nospy` - clear spypoints

`nospy(pred)`

ARGUMENT

pred	: atom or list of atoms

USE

The *pred* argument may be either a single relation name or a list of relation names. The call clears spypoints from the specified relations.

13.7 `nospyall` - clear all spypoints

`nospyall`

USE

This call clears all spypoints currently set.

13.8 `trace` - set tracing on

`trace`

USE

Tracing of interpreted relations is switched on. (See the Environment Guide for a description of tracing of programs.)

13.9 `notrace` - switch tracing off

`notrace`

USE

This call switches tracing off.

13.10 unknown - set action on undefined relations

`unknown(old_action,new_action)`

ARGUMENTS

`old_action`	: variable, previous action
`new_action`	: atom, new action

USE

When an undefined predicate is encountered during an evaluation there are different ways of dealing with it (provided it is not *dynamic* - see the `dynamic` primitive below).

The *`new_action`* argument specifies what action should be taken when an unknown relation name is encountered, and should be one of

```
fail
trace
true
```

The `fail` option will cause the unknown call to fail.
The `trace` option will cause the error handler to be called (see the chapter on Implementing an Application).
The `true` option will make the unknown call succeed.

The *`old_action`* argument returns the currently defined action procedure.

The call

```
unknown(Action, Action)
```

will simply return the current action without changing it.

13.11 dynamic - declare dynamic relations

`dynamic(pred)`

ARGUMENTS

`pred`	: atom or list of atoms

USE

This call is used to over-ride the `unknown` action for the relation or relations specified by the *`pred`* argument. If an undefined `dynamic` relation is encountered it will automatically fail.

13.12 **`ticks`** - return the system elapsed time, or wait for a time

`ticks(time)`

ARGUMENT

`time` : variable, or non-negative integer

USE

If *`time`* is a variable it will be bound to an integer which is the number of 60ths of a second that have elapsed since the system was booted up. It may be used to time the amount of processor time taken by certain calls.

If *`time`* is an integer, then program execution is suspended for *`time`* 60ths of a second.

(This timing mechanism is not totally reliable since heavy use is made of interrupts within the Macintosh, and time taken for processing interrupts is not removed from the system clock.)

14 Serial I/O

The following primitives give access to the printer and modem serial ports. Once a channel has been opened and configured, data may be read from and written to the ports using the general I/O primitives `read`, `write` etc. specifying the channel to be either `printer` or `modem`.

WARNING: There are some known problems associated with use of the printer port; for example there may be interference with Appletalk, or the printer port may not work at all, depending on your version of the operating system.

14.1 **`seropen`** - open a serial channel

`seropen(`*`channel`*`)`

ARGUMENTS

`channel` : atom

USE

This primitive opens the specified serial channel.
The *`channel`* argument should be either `printer` or `modem`.

14.2 **`serconfig`** - configure a serial channel

`serconfig(`*`channel,port,baudrate,databits,parity,stopbits`*`)`

ARGUMENTS

`channel`	: atom
`port`	: atom
`baudrate`	: integer
`databits`	: integer
`parity`	: atom
`stopbits`	: number

USE

This call configures the specified serial channel.

The *`channel`* argument should be either `printer` or `modem`.

The *`port`* argument must be one of the atoms:

`in` `out` `both`

depending on whether the channel is to be used for input, output or both input and output.

The *`baudrate`* argument is the baud rate setting for the channel.

It must have one of the following integer values:

`300 600 1200 1800 2400 3600 4800 7200 9600 19200 57600`

NOTE: The maximum baud rate for the printer port is 300 for input, and 9600 for output.

The *`databits`* argument is the number of data bits per character, which should have the value `5`, `6`, `7` or `8`.

The *`parity`* must be one of the atoms:

`odd` `even` `none`

The *`stopbits`* argument is the number of stop bits. It should have the value `1`, `1.5` or `2`.

14.3 `serstatus` - get the status of a serial channel buffer

`serstatus(`*`channel,port,count`*`)`

ARGUMENTS

`channel`	: atom
`port`	: atom
`count`	: variable or integer

USE

If *`count`* is a variable, this call is used to find the number of characters remaining in the serial buffer.
If *`count`* is an integer, the call will succeed if there are `count` characters remaining in the serial buffer.
The *`channel`* argument must be either `printer` or `modem`.
The *`port`* argument must be either `in` or `out`.

14.4 `serxonxoff` - enable Xon/Xoff flow control

`serxonoff(`*`channel`*`)`

ARGUMENT

`channel`	: atom

USE

This call enables Xon/Xoff data flow control (the Xon and Xoff characters are Control-Q and Control-S respectively).
The *`channel`* argument should be either `printer` or `modem`.

14.5 `serets` - enable CTS hardware handshake

`serets(`*`channel`*`)`

ARGUMENT

`channel`	: atom

USE

This call enables CTS hardware handshake.
The *`channel`* argument should be either `printer` or `modem`.

15 Menu Handling

MacPROLOG has several primitives that enable you to create, manipulate and delete the pull-down menus of the menu bar at the top of the Mac display.

If you add a new pull-down menu with name **Name**, `Name` *must* also be defined as a *unary* relation. When the user uses the mouse to select an item **Item** from this menu, this is equivalent to calling the relation `Name` with argument `Item`, i.e. it is equivalent to the call

```
'Name'('Item')
```

If `Name` is not defined there will be an error message to the effect that the menu is not defined. The invoked `Name` program can be interpreted or compiled. It can display dialogues, create windows, or even change the menu bar. It can ultimately succeed or fail. At the end of the evaluation the user can again select a command from the menu bar.

All the menus of the programming environments, except the **File** and menus, are implemented as MacPROLOG programs.

15.1 `install_menu` - create a new pull down menu

`install_menu(`*`name,item_list`*`)`

ARGUMENTS

name	:	atom, name of menu
item_list	:	list of atoms, menu items

USE

Any existing pull-down menu called *name* is deleted and a new pull-down menu is added to the menu bar to the right of the current rightmost menu.The *item_list* contains the items to appear in the menu.

The *item_list* can either contain a single quoted atom with the menu items separated by semi-colons, or it can contain the items as individual atoms (see the example below).

The menu item names can contain extra control information as prefix or postfix character sequences.

A quoted atom name ending with / followed by a capital letter associates a control key with that menu item. (The Macintosh 'control' key is the one marked ⌘ on the keyboard.) For example,

```
'Item/G'
```

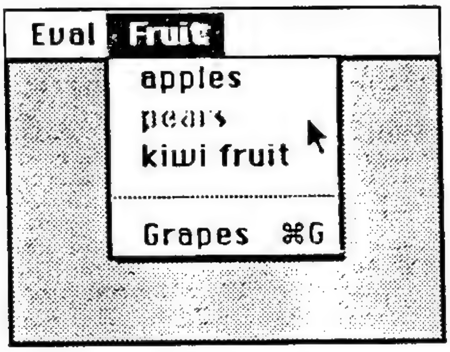
will associate with the menu item name `Item` the control key ⌘ G. Typing ⌘ G will then be equivalent to selecting that menu item. You should make sure that the control characters associated with different menu items are distinct. (But if you assigned ⌘ G to a second menu item the `/G` would be ignored for that second item.)

If a quoted atom name is prefixed by a round bracket (then the item will initially be disabled, i.e. it will appear in grey and cannot be selected by the user. (See also the primitives `enable_item` and `disable_item`.)
To insert a horizontal separating bar between two items in the menu, separate the item names in *`item_list`* with the atom `'(-'`.

EXAMPLE

The following call will create a four item **Fruit** menu with **pears** initially disabled, and a horizontal separating bar between **kiwi fruit** and **Grapes**. Pressing ⌘ G will be equivalent to selecting **Grapes** from the menu.

```
install_menu('Fruit',
     [apples,'(pears','kiwi fruit','(-','Grapes/G'])
```



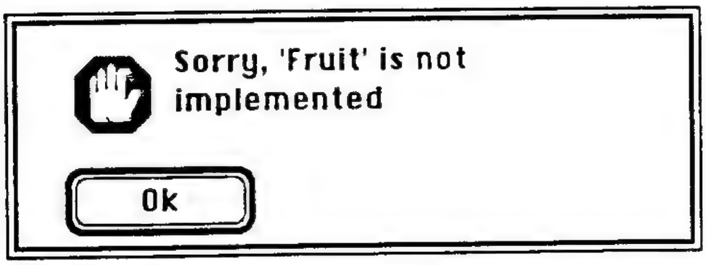
This is also equivalent to the call

```
install_menu('Fruit',['apples;(pears;kiwi fruit;(-;Grapes/G'])
```

Remember that there must be a program of the following form defining `'Fruit'`:

```
'Fruit'(apples):-
   /* clause for apples option */

'Fruit'('kiwi fruit'):-
   /* clause for kiwi fruit option */
```

and similarly for `pears` and `Grapes`, otherwise an error will be generated when one of the menu items is selected:



15.2 `kill_menu` - delete a menu from the menu bar

`kill_menu(`*`name`*`)`

ARGUMENT

name : atom, name of a menu

SIDE-EFFECT

The menu *name* is deleted from the top menu bar. The associated program which defines *name* is not deleted.

PRAGMATICS

You can use this primitive to delete the menus of the programming environments as well as your own menus. However, do not delete the menu or the **File** menu.

15.3 `clear_menu` - remove all items from a menu

`clear_menu(`*`name`*`)`

ARGUMENT

name : atom, name of a menu

SIDE-EFFECT

This will erase all the entries in the menu, but leave the menu name on the menu bar. It may be used before inserting a new set of entries for a menu.

15.4 `disable_menu` - disable a menu

`disable_menu(`*`menu`*`)`

ARGUMENTS

menu : atom, name of a menu

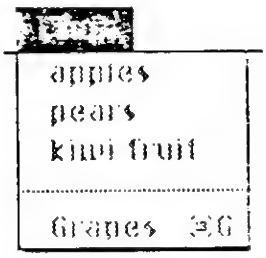
USE

The whole menu *menu* is disabled. The menu name appears 'greyed out' in the menu bar, and all its items also appear in grey if the user pulls down the menu. Individual items in a disabled menu may still be disabled or enabled with `disable_item` or `enable_item`, but enabled items cannot be selected by the user until the menu itself is enabled.

EXAMPLE

The **Fruit** menu described at the beginning of the chapter may be disabled by the call

```
disable_menu 'Fruit'
```



No item from this menu may now be selected.

15.5 `enable_menu` - enable a previously disabled menu

`enable_menu(`*`menu`*`)`

ARGUMENTS

menu : atom, name of a menu

USES

The whole menu *menu* is enabled. Its name appears in black on the menu bar, and all items (except those individually disabled by `disable_item`) can now be selected.

15.6 **`extend_menu`** - extend a current list of menu items

`extend_menu(`*`menu, items`*`)`

ARGUMENTS

menu	: atom, name of a menu
items	: list of atoms, new menu items

USE

The list of items of *menu* is extended below by the extra *items*. The menu item names have the same form as for `install_menu`.

15.7 **`disable_item`** - disable a menu item

`disable_item(`*`menu, item`*`)`

ARGUMENTS

menu	: atom, name of a menu
item	: atom, name of menu item

USE

The *item* of *menu* is disabled. It cannot now be selected. The disabled item appears 'greyed out' in the menu and is not highlighted when the user moves the mouse over the item.

15.8 **`enable_item`** - enable a previously disabled item

`enable_item(`*`menu, item`*`)`

ARGUMENTS

menu	: atom, name of a menu
item	: atom, name of menu item

USES

The *item* of *menu* is enabled. It can now be selected by the user, and will appear in black on the menu display.

15.9 `mark_item` - mark a menu item

```
mark_item(menu,item)
mark_item(menu,item,code)
```

ARGUMENTS

menu	: atom, name of a menu
item	: atom, name of menu item
code	: integer, ASCII code of mark character

USE

1. Two argument call The *item* of *menu* is marked with a tick. Such a marking can be used to indicate the setting of some global flag. Whether or not an item is marked can be tested with the `marked_item` primitive.

2. Three argument call If an integer is given as a third argument, the integer is interpreted as an ASCII code and that character is used instead of a tick to mark the menu item.
Some codes you might like to use are:

17 - ⌘
19 - ◆
20 -

For example, the items **apples** and **kiwi fruit** may be marked using the two calls:

```
    mark_item('Fruit',apples,20)
and mark_item('Fruit','kiwi fruit')
```

15.10 `unmark_item` - unmark a marked menu item

```
unmark_item(menu,item)
```

ARGUMENTS

menu	: atom, name of a menu
item	: atom, name of menu item

USE

The previously marked *item* of *menu* is unmarked - the tick (or other mark character) is removed.

15.11 `marked_item` - test if a menu item is marked

`marked_item(menu,item)`

ARGUMENTS

menu	: atom, name of a menu
item	: atom, name of menu item

USE

It is used to test if the *item* of *menu* is currently marked. If it is the call succeeds, otherwise the call fails.

15.12 `rename_item` - rename a menu item

`rename_item(menu,item,new_item)`

ARGUMENTS

menu	: atom, name of a menu
item	: atom, name of menu item
new_item	: atom, new name for menu item

USE

The *item* of *menu* is renamed *new_item*.

15.13 **`style_item`** - change the style of a menu item

`style_item(menu,item,style)`

ARGUMENTS

menu	: atom, name of a menu
item	: atom, name of menu item
style	: atom, name of an item style

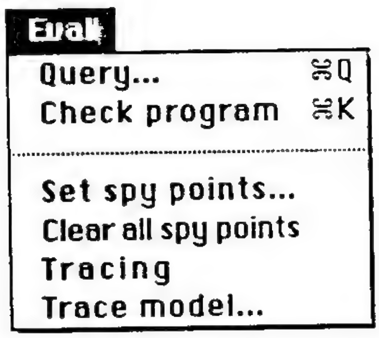
USE

The *item* of *menu* will be displayed in the designated style. The allowed values for the `style` argument are the atoms:

```
normal
bold
italic
underline
shadow
condense
extend
```

For example, the following program would change the Eval menu as shown below.

```
change_eval:-
    style_item('Eval','Clear all spy points',condense),
    style_item('Eval','Tracing',extend).
```



PRAGMATICS

Styles can be used to give status indication to the user.

16 Predefined Dialogues

Dialogue windows may be used to give information or to accept input from users during the running of a program. You can build your own dialogue windows, or use some of the predefined dialogues of MacPROLOG. This chapter describes the predefined dialogues available; see the chapters on Dialogue Building and Advanced Dialogues if you wish to build your own.

There are two types of dialogues: *Modal* and *Modeless.*

A *modal* dialogue must be answered before anything else is done. You cannot select another window or pull down a menu whilst a modal dialogue is displayed. You must respond to the dialogue.

A *modeless* dialogue does not need to be immediately answered. You can select some other window, scroll it, edit it, and paste text into the modeless dialogue edit fields. You can also execute any command of the pull down menus before responding to the dialogue. This means that you can even start a second MacPROLOG process. The process that created the modeless dialogue will be suspended until you select the dialogue and click its **Ok** or **Cancel** button. At that point the suspended process is resumed. Modeless dialogue windows may also be moved around the screen.

Keyboard input during an evaluation can be achieved using primitives that display dialogues containing a prompt message and an edit field into which the users may type. On hitting *Return* or clicking the **Ok** button of the dialogue, the typed input is bound to the variable of the invoking call. Clicking the **Cancel** button of the dialogue causes the call to fail.

The primitives `prompt_read` and `prompt_gread` read arbitrary terms. These two display modeless dialogues.

The primitive `ask` invokes a modeless dialogue which accepts arbitrary user input without performing any syntax checks.

The primitives `yesno` and `myesno` invoke dialogues which display a message that requires only a *yes/no* answer via dialogue buttons. `yesno` creates a modeless dialogue, `myesno` a modal dialogue.

There is also the primitive `scroll_menu` which displays a scrolling menu of a supplied list of atoms from which one or more items may be selected with the mouse. This is a modal dialogue.

There are three special purpose dialogue primitives for giving warnings and messages - `message`, `warning`, and `errormessage`.

Another primitive, `banner`, provides a non-interactive dialogue for the display of information to the user, which will terminate on the completion of a specified goal. This can be used to give information to the user during time-consuming processes such as loading large files.

An online help facility may be implemented using the `help` primitive. This reads text from a help file and displays it on the screen in a dialogue.

16.1 `prompt_read` - prompted read of hollow term

`prompt_read(`*`prompt_list, term`*`)`

ARGUMENTS

prompt_list	: a list of terms
term	: a variable, will be the term read in

USES

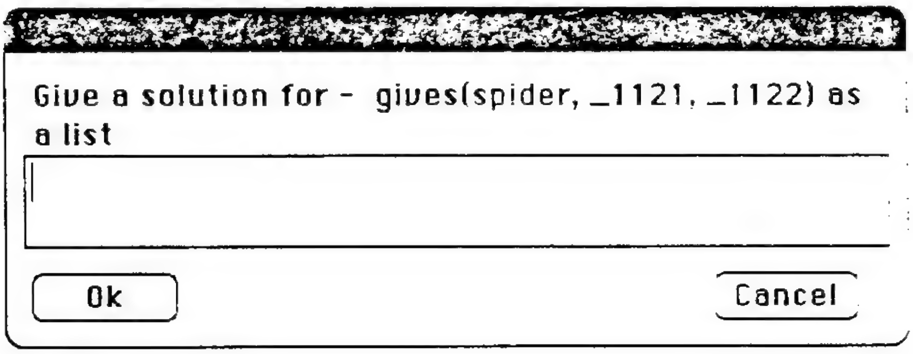
A modeless dialogue is displayed with an edit field into which the term must be entered. The prompt message comprises the sequence of terms in *prompt_list* separated by spaces. In other words, *prompt_list* will be displayed without the outer list brackets and without the separating commas between the terms on the list. Also atoms will not be quoted.
The characters entered into the edit field of the dialogue will be read in as a term. A terminating full stop is not needed but can be entered.

Variable names in the entered term are converted into variables. The entered term becomes the binding of the *term* argument when the *Return* key is hit or the **Ok** button of the dialogue is selected. Selecting the **Cancel** button causes the call to fail.

The call

```
prompt_read(['Give a solution for - ',
                    gives(spider,X,Y),as,a,list],Z)
```

will generate the dialogue:



(the variable names will be different). If the user types

```
[fright,'Miss Muffet']
```

into the edit field of the dialogue then the variable Z of the call will be bound to the list `[fright,'Miss Muffet']`.
Note that if the user simply presses *Return* or clicks on the **Ok** button without typing anything, then the variable Z will be bound to the atom `end_of_file`.

If there is a syntax error in the edit field, an error dialogue will be displayed - see `prompt_gread`.

16.2 **prompt_gread** - prompted read of ground term

prompt_gread(*prompt_list, term*)
prompt_gread(*prompt_list, term, varnames*)

ARGUMENTS

prompt_list	: a list of terms
term	: variable, will be read-in term
varnames	: variable, will be variable names of the read-in term

USE

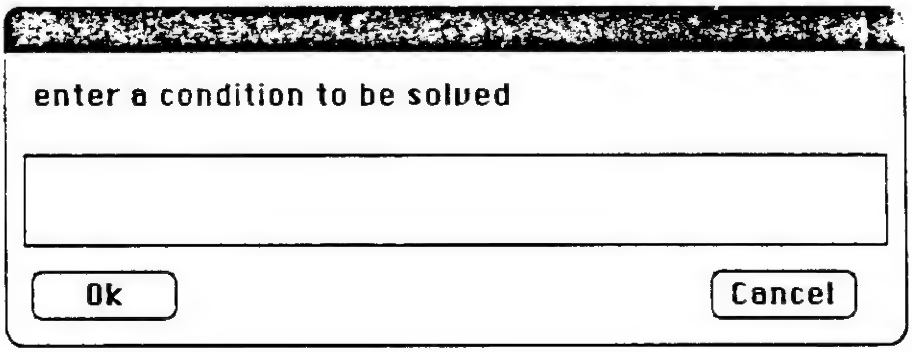
The two argument form has exactly the same use as `prompt_read` except that variable names in the read-in term are left as atoms. They are not converted into variables. (They will be displayed as quoted atoms if you `writeq` the term.)

The three argument form also displays the same dialogue and reads in ground terms. The difference is that the third argument will be bound to the list of variable names of the read-in term.

The call

```
prompt_gread(['enter a condition to be solved'],Gterm,Varnames)
```

will display the dialogue



If the response is to enter

```
likes(keith,People)
```

into the edit field, then `Gterm` will be bound to `likes(keith,'People')` and `Varnames` will be bound to `['People']`.

PRAGMATICS

Use the two argument form of this primitive instead of `prompt_read` when you know that the input sequence of terms will not need to contain variables and you do not want the user to have to remember to quote upper case names.

Use the three argument form when you want to remember the variable names of the read-in terms. You can always convert the read-in term into a hollow term using the `tohollow` primitive. For example, after the above `prompt_gread` call, the call

```
tohollow(Gterm, Hterm, Varnames)
```

will result in `Hterm` becoming `likes(keith,_123)` in which `_123` is a real variable which can be bound by a unification.

By using `prompt_gread` and then `tohollow` rather than `prompt_read` you have retained the variable name used in the call which can be used when the solution is displayed.

ERRORS

If the user clicks on the **Ok** button and there is a syntax error in the edit field, a modal dialogue will be displayed at the bottom of the screen:

The user must then click the **Ok** button on this dialogue and return to the `prompt_gread` (or `prompt_read`) dialogue to either correct the error or cancel the dialogue.

16.3 ask - prompted read of arbitrary input

ask(*prompt_list,input*)

ARGUMENTS

prompt_list	: a list of terms
input	: variable, will become a list of tokens

USE

To accept input from the keyboard as a sequence of atomic terms. No syntax checks are made on the input.

A modeless dialogue is displayed with an edit field, **Ok** and **Cancel** buttons, and a message formed from the *prompt_list* argument. The terms of *prompt_list* are displayed as a sequence of terms separated by spaces, as with `prompt_read`.

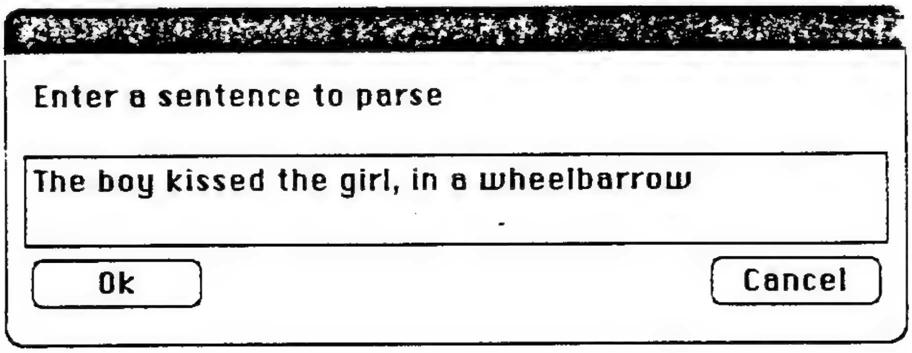
The *input* argument will be bound to a list of the 'tokens' entered in the edit field of the dialogue. A token is any symbol, word, separator, etc. (for more information about tokens see the `edintok` primitive).

EXAMPLE

The call

```
ask(['Enter',a,sentence,to,parse],List)
```

will display a dialogue with an empty edit field into which the user types text.



If the user enters the above text, then `List` will be bound to the list of tokens

```
['The',boy,kissed,the,girl,',',in,a,wheelbarrow]
```

The calling PROLOG program may then process this text in an appropriate way (for example all punctuation may be ignored).

The `ask` primitive is useful when you are not necessarily expecting valid Edinburgh syntax terms.

16.4 yesno - display a prompt that requires a yes/no answer

`yesno(prompt_list)`

ARGUMENTS

`prompt_list` : a list of terms

USE

To get a yes or no answer to a question. `yesno` displays a modeless dialogue with **Yes** and **No** answer buttons and a message formed from the *`prompt_list`* argument.

The sequence of terms of the *`prompt_list`* argument is displayed as a sequence of terms separated by spaces, as with `prompt_read`.

The only allowed response is selecting the **Yes** or the **No** button on the dialogue or hitting the *Return* key.

If the **Yes** button is selected or *Return* is hit the call succeeds.

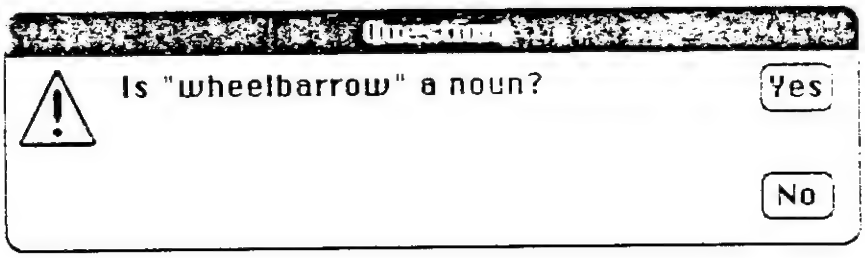
If the **No** button is selected the call fails.

EXAMPLE

The call

```
yesno(['Is "wheelbarrow" a noun?'])
```

displays the dialogue



16.5 **myesno - display a prompt that requires an immediate yes/no answer**

myesno(*prompt_list*)

ARGUMENTS

prompt_list : a list of terms

USE

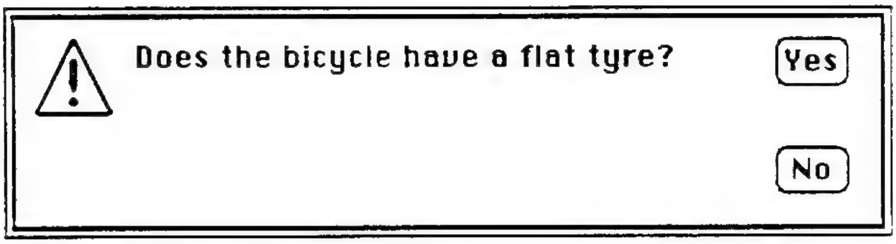
Exactly the same as yesno, except that the dialogue is modal and so the user must respond by clicking on either the **Yes** or **No** answer button on the dialogue before doing anything else. Any attempt to select another window or pull down a menu before responding to the dialogue will cause the bell to sound.

EXAMPLE

The call

```
myesno(['Does the bicycle have a flat tyre?'])
```

will display the following dialogue.



16.6 **`scroll_menu`** - display a scrolling menu for item selection

```
scroll_menu(prompt_list,menu_list,
                    preselected,selected)
```

ARGUMENTS

prompt_list	: a list of terms
menu_list	: a list of atoms
preselected	: a list of atoms
selected	: a variable

USE

To allow selection of one or more items from a given list. A modal dialogue containing a scolling menu is displayed. It must be responded to before anything else is done.

The scrolling menu dialogue is displayed with the *menu_list* atoms as the menu items and with the sublist of the *preselected* atoms already selected (i.e. in reverse video). Using the mouse click and shift-click, the list of selected menu items can be changed. No item need be selected.

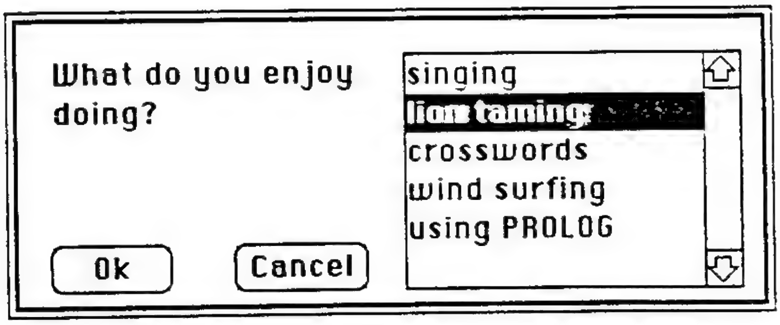
On hitting the *Return* key or upon selecting the **Ok** button of the dialogue the call succeeds with the *selected* variable bound to the list of selected items in the menu. On selecting the **Cancel** button the call fails.

As with the other dialogues, the *prompt_list* will be displayed as a sequence of terms separated by spaces, and atoms will not be quoted.

The call

```
scroll_menu(['What do you enjoy doing?'],
      [singing,'lion taming',crosswords,
      'wind surfing','using PROLOG'],
      ['lion taming'], S)
```

will result in this scrolling menu being displayed:



If the selection is extended by shift-clicking on `using PROLOG` before the **Ok** button is selected then `S` will be bound to the list `['lion taming','using PROLOG']`.

16.7 **message - display a modal dialogue with a message**

`message(term_list)`

ARGUMENTS

`term_list` : list of terms to be displayed

USE

A modal dialogue is displayed on the screen. The terms in the `term_list` will be displayed inside the dialogue as a sequence of terms separated by spaces, and with no outer list brackets.

The dialogue also contains an **Ok** button which the user must click on before continuing.

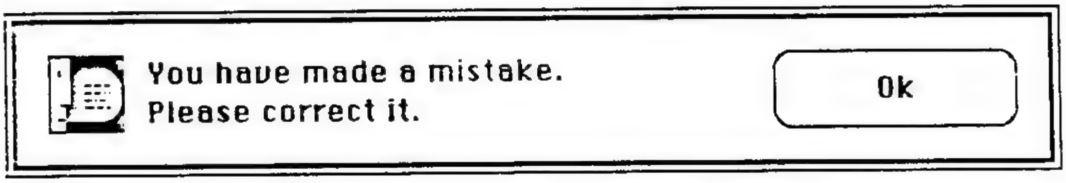
Notice that `message` is purely for the display of information, since the user is not given any options. The only course of action available is to click the **Ok** button.

EXAMPLE

The call

```
message(['You have made a mistake. ~MPlease correct it.'])
```

displays the modal dialogue



Note the use of the ~M in the *prompt_list* argument to denote a carriage return.

16.8 `warning` - display a modal warning dialogue with a message

`warning(term_list)`

ARGUMENTS

`term_list` : list of terms to be displayed

USE

A modal dialogue is displayed on the screen. The terms in the `term_list` will be displayed inside the dialogue as a sequence of terms separated by spaces, and with no outer list brackets.

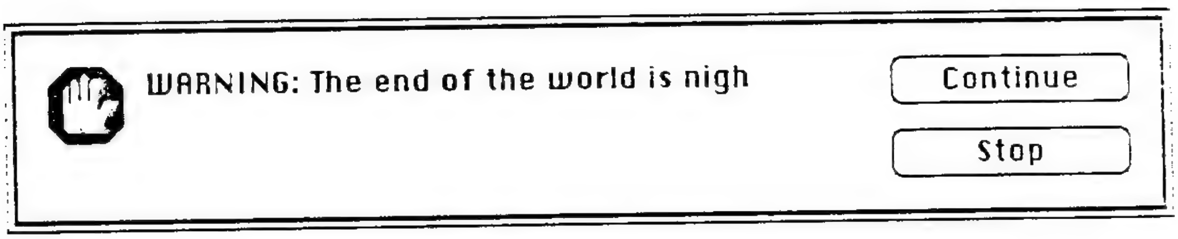
The dialogue also contains a **Continue** button and a **Stop** button. If the user clicks on the **Stop** button the current process terminates; if the user clicks the **Continue** button the process continues.

EXAMPLE

The call

```
warning(['WARNING:','The end of the world is nigh'])
```

will display the modal dialogue



16.9 **errormessage** - display a modal error dialogue with a message

errormessage(*term_list*)

ARGUMENTS

term_list	: list of terms to be displayed

USE

A modal dialogue is displayed on the screen. The terms in the `term_list` will be displayed inside the dialogue as a sequence of terms separated by spaces, and with no outer list brackets.

The dialogue also contains a **Stop** button which the user must click on. This terminates the current process.

EXAMPLE

The call

```
errormessage(['The end of the world has come'])
```

will display the following dialogue.

16.10 **banner** - display information during a process

banner(*call*, *message*)

ARGUMENTS

call	: term representing the goal to be executed
message	: term list, the message to be displayed

USE

This primitive displays a non-interactive dialogue box on the screen (it has no buttons). The *message* list will be displayed in the usual way (i.e. as terms separated by spaces with no outer list brackets). The *call* is then executed, and when the evaluation is completed the dialogue terminates. A call to `banner` always succeeds, regardless of whether or not *call* succeeds or fails.

For example, the following call will give an information banner whilst loading a file.

```
banner(consult('Large File'),
        ['Please wait - loading','Large File'])
```

16.11 beep - sound the bell

beep(*time*)

ARGUMENT

time : integer

USE

This primitive will cause the system to beep for the specified `time` (in 60ths of a second). The volume of the beep depends on the current speaker volume setting, which the user can set from the Control Panel desk accessory. If the speaker volume has been set to 0, `beep` will cause the Menu Bar to blink once.

16.12 help - online help facility

help(*file*, *subject*, *title*, *text*)
help(*file*, *subject*, *title*)

ARGUMENTS

file	: atom, the name of an ASCII text file
subject	: atom, a main entry in the help file
title	: atom or variable. A sub-entry in the `subject`
text	: variable. Becomes the help associated with `subject` and `title`

USE

This call is used to provide an online help facility. The information to be displayed is stored in the ASCII text file `file`, which must be in the current default volume when `help` is first called.

1. Four argument use
`text` will be bound to the atom (up to 255 characters) corresponding to the entry for `subject` and `title` in the help file named by `file`.

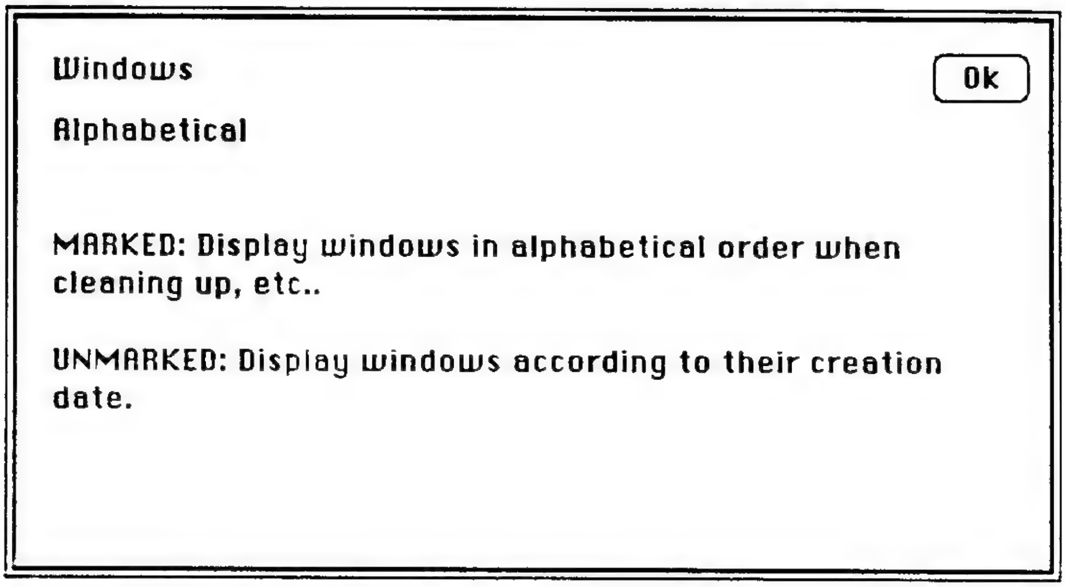
If `title` is not given and there are multiple entries for `subject` then there will be multiple solutions to the call.

2. Three argument use
Similar to the four argument call, but instead of returning the help constant as the value of an argument, the `subject`, `title` and help text are displayed in a modal dialogue with a single **Ok** button.

For example, in the MacPROLOG programming environment the call

```
help('Menu Help','Windows','Alphabetical')
```

will display the following dialogue



The on-line help for pull-down menus in the MacPROLOG programming environment is implemented by programs such as

```
'Search'(C):- recall(help,on),!,
              help('Menu Help','Search',C).

'Search'('What to find ...'):-
              /* program for this menu item */
```

and so on.

The first clause above tests to see if help mode is on. You can define your own menu programs in the same way, accessing your own help file.

SIDE EFFECTS

The initial call to `help` for some *`file`* will cause the complete contents of the file *`file`* to be loaded into an invisible display window with the same name. It is actually this display window which is used as the source of the help information.

For users who wish to make use of this facility in their own applications, there is a simple structure for the help file. Each entry in the file specifies the subject and title of the help item, followed by the text for this item enclosed in comment brackets `/*` and `*/`, i.e. an entry is of the form

```
subject title /* ... text for this item ...*/
```

The `title` may be omitted:

```
subject /* ... text for this item ... */
```

which is equivalent to

```
subject '' /* ... text for this item ... */
```

NOTE:

a) It is important that there are spaces on either side of the `/*` and `*/` in the help file. If there are no surrounding spaces, `/*` or `*/` will be not be correctly recognised as delimiters of the text.

b) Different entries for the same `subject` do not have to be contiguous in the help file. The display window is simply searched for matching text; it is not compiled.

17 Dialogue Building

There are two powerful primitives that enable you to define, display and extract information from dialogues. `dialog` is for labelled modeless dialogues and `mdialog` is for unlabelled modal dialogues. See the chapter on Predefined Dialogues for a description of the differences between the two types of dialogues.

Both primitives allow you to define and use dialogues with icons, text fields, edit fields, check boxes, radio buttons and ordinary buttons. In addition, one scrolling menu is allowed in a modal dialogue.

MacPROLOG also allows picture items in dialogues. For a description of these, and of other special dialogue handling features, see the chapter on Advanced Dialogues.

17.1 dialog - display and read from a labelled modeless dialogue

dialog(*title*, *down_pos*, *in_pos*, *depth*, *width*,
\t*format_list*, *button*)

ARGUMENTS

title	: atom, label displayed at top of dialogue
down_pos	: integer, position of top of dialogue as a number of pixels down from the top of the Mac display. This should be at least 40 to avoid the menu bar. The overall depth of the Mac Plus display is 342 pixels.
in_pos	: integer, position of left of dialogue as a number of pixels in from the left of the Mac display. The overall width of the Mac Plus display is 512 pixels.
depth	: integer, depth of the dialogue in pixels
width	: integer, width of the dialogue in pixels
format_list	: list of dialogue field format descriptors - see below
button	: variable, will become an integer indicating which button was selected

USE

A call to this primitive displays a dialogue of the size and position specified with edit, button and other fields as specified by the *format_list*. The dialogue has rounded corners and the title is displayed in a title bar across the top of the dialogue.

When the user selects one of the buttons on the dialogue - displayed as oval fields - the number of the button selected will be returned as the value of the variable *button*. This number is the position of the format descriptor of the selected button within the *format_list*.

As a general rule, the first two items in the format list should be button descriptors. The first item should be an **Ok** button, corresponding to the normal response to the dialogue. The second item should be a **Cancel** button, corresponding to the user's wish not to respond to the dialogue. If the **Cancel** button is selected, that is if the value that would be returned for `button` is 2, then the call to `dialog` fails, and *none* of the other dialogue fields are read.

If the user hits ⌘. (the ⌘ key with the full stop key) this is also a cancel action and it also causes a failure of the call. If the second format item is not a button this is the only way to cancel the dialogue and fail the call.

If any button other than the **Cancel** button is selected, that is any button with a format descriptor which is *not* the second descriptor on the format list, and there are no syntax errors in the edit fields of the dialogue, the call succeeds. The value of `button` tells you the button that has been selected. All the other fields in which the user might have entered information, such as check buttons, radio buttons and edit fields, have their values read in to variables supplied as part of their format description in the format list.

If the user hits *Return*, this is also interpreted as a normal response to the dialogue and the value of `button` will be 1. Again, the other field values will be read in to the variables of the format descriptors, providing there are no syntax errors in the edit fields.

If there are syntax errors in an edit field of a dialogue, a modal error dialogue will be displayed. The syntax errors must then be corrected (or the dialogue cancelled).

THE FORMAT DESCRIPTORS

The field format descriptors are all terms of the form

```
field_type(down_pos,in_pos,depth,width,...)
```

where

`field_type`	is an atom giving the type of the field
`down_pos`	is an integer giving the relative position of the top of the field as a number of pixels down from the top of the dialogue. This should be at least 40 to avoid the title bar.
`in_pos`	is an integer giving the relative position of the left of the field as a number of pixels in from the left of the dialogue
`depth`	is an integer giving the depth of the field in pixels. This should be at least 20 for fields that will hold text. A line of text is 16 pixels high.
`width`	is an integer giving the width of the field in pixels. For fields that will hold text allow 10 pixels for each character.

and the ... extra parameters depend on the type of the field.

Where the type is a field from which values will be read the last parameter is always a variable or a term containing variables into which the field values will be read.

The allowed field descriptors for a modeless dialogue are as follows.

1. **`edit(`*`down_pos, in_pos, depth, width, initial_value, final_value`*`)`**

This creates an edit field within the dialogue of the size and position specified. The edit field is displayed as a rectangular box. The *`initial_value`* parameter is a term giving the display format and value of the initial contents of the edit field: the contents that the user will be able to edit. The *`final_value`* is a term indicating the read back format of the final contents of the edit field. The *`final_value`* format term must contain one or more variables to hold the read-in values.

The ability to prefill an edit field is extremely useful. It enables you to build an application in which the user's entered values are anticipated with default values. You can use the interpreted data base or the property management primitives to remember a default value, which might be the value entered into the edit field when the dialogue was last displayed. The read back format does not need to be the same as the prefill format.

The allowed prefill formats for the *`initial_value`* term are:

`i_atom`	*`i_atom`* is an atom. The atom will be displayed in the edit field unquoted. This enables you to preload an edit field with a sequence of characters and spaces given as a quoted atom. To leave the edit field empty, use the empty quoted atom `''`.
`writeq(`*`iterm`*`)`	*`iterm`* is any term. The term will be displayed in the edit field as though it were written with `writeq`.
`writeq(`*`iterm, vns`*`)`	*`iterm`* is any ground term, *`vns`* is a list of quoted atoms that are usually variable names in *`iterm`*. The term will be displayed in the edit field as though it were written with `writeq` except that atoms in *`vns`* will not be quoted. This is the format you should use to display user inputs that have previously been read in from the edit field using the `gread` format.
`bytes(`*`bytes`*`)`	*`bytes`* is a list of byte codes. The list of bytes will be displayed concatenated as a sequence of characters. For example, `bytes([65, 66, 67])` will be displayed as the text ABC.
`tokens(`*`toks`*`)`	*`toks`* is a list of `token` terms (see the description of the use of `tokens` as a read back format). The main use of this prefill format is to redisplay in an edit field a sequence of tokens read in from a previous use of the dialogue.
`words(`*`wds`*`)`	*`wds`* is a list of atomic terms The list will be displayed in the edit field as a sequence without the surrounding list brackets and without the separating commas. The items on the list will be separated by spaces.

The allowed read back formats for the `final_value` term are:

`f_atom` — `f_atom` is a variable. The final character sequence contents of the edit field will be read back in as a single atom which will become the value of the variable *f_atom*. If need be, you can then parse this by using `name` to map the atom into a list of bytes.

`read(fterm)` — *fterm* is a variable. The final character sequence contents of the edit field will be read back in as a term. Variable names in the term are replaced by variables.

`gread(fterm, vns)` — *fterm* and *vns* are variables. This is exactly the same as `read(fterm)` except that variable names in the read-in term are left as atoms and the list of all the variable names in the read-in term are returned as the binding for *vns*. The read-in term can be subsequently converted to a hollow term using `tohollow`. See the description of `prompt_gread` for an example of the possible use of this format descriptor.

`bytes(bytes)` — *bytes* is a variable. It will be bound to a list of the ASCII codes of the sequence of characters entered in the edit field. For example, if the entered text is `ABC` *bytes* will be bound to the list `[65, 66, 67]`.

`tokens(toks)` — *toks* is a variable. This will be bound to a list of terms of the form `token(token, type)` where *token* is an Edinburgh syntax token and *type* is the integer identifier of the token type. The possible token types are:

0 Separator (' ' or '. ' space or dot space)
1 Punctuation (either '(' ')' '[' ']' '|' '{' '}' or ',')
2 Single character symbol ('!' ';')
3 Symbolic name (sequence of graphic characters e.g. ££)
4 Uppercase name (alphanumeric starting with uppercase letter or '_')
5 Lowercase name
6 Number (only positive numbers)
7 String
8 Quoted atom

In effect, this format descriptor causes the entered text to be tokenised and returned as a list of the entered sequence of tokens.
For example, if the entered text is

```
[mary,4,!]
```

toks will be bound to the list of terms

```
[token('[',1), token(mary,5), token(',',1),
token(4,6), token(',',1), token(!,2),
token(']',1)].
```

`words(wds)` *wds* is a variable. This will be bound to a list of the tokens of the edit field. This is the same list of tokens as returned by `tokens` except that the token type is not given.
For example, if the entered text is

```
clare, dave and brian like cats
```

wds will be bound to the list of tokens

```
[clare,',',dave,and,brian,like,cats]
```

Use `words` if you want to access the entered text as a list of atoms. No syntax checks will be made on this input.

2. **text(*down_pos, in_pos, depth, width, text_value*)**

This creates a text field within the dialogue box at the position specified. The *text_value* gives the display format and value for the text to be displayed. It will appear without a surrounding rectangle.

The allowed formats for *text_value* are the same as the prefill formats for an edit field with the addition of:

`wseq(iterm_list)` *iterm_list* is a list of terms. The terms in the list will be displayed in the text field as a sequence of terms separated by spaces (not commas) and with no outer list brackets. Each term is displayed as though it were written with `write`. For example, if the text format is

```
wseq(
     ['Is the following condition true:',
      likes(keith,mary),?])
```

then the contents of the text field will be

```
Is the following condition true: likes(keith,mary)?
```

3. **button(*down_pos, in_pos, depth, width, label*)**

This creates a button field within the dialogue of the size and position specified. The button field is displayed as a rectangular box with rounded corners. The *label* must be an atom, which is displayed in the button field. A button is selected using the mouse.

Unlike the other data entry fields, the value of the selected button is not returned by the binding of a variable given in the format descriptor. This is because clicking on a button usually terminates the dialogue. The value of the selected button is returned as the binding for the last *button* argument of the `dialog` call. This value is the position of the format descriptor for the selected button within the entire format list of the `dialog` call. Note the comments above about the special role for **Ok** and **Cancel** buttons.

(See the chapter on Advanced Dialogues for a description of the extended `dialog` call in which clicking on a button may *not* necessarily terminate the dialogue.)

4. **check(*down_pos,in_pos,depth,width,label, initial_value,final_value*)**

This creates a check box field within the dialogue, of the size and position specified. The *depth* and *width* size of the field must be large enough for the check box and its adjacent label which is written to the right of the box. The `label` is any atom.

The *final_value* must be a variable. It will be bound to on or `off` indicating whether or not the check box was in the selected state when the dialogue was read.

The *initial_value* is one of:

`on` The check box is displayed initially selected with a cross in the box.

`off` The check box is displayed initially unselected. It is an empty box.

5. **radio(*down_pos,in_pos,depth,width,label, initial_value,final_value*)**

This creates a radio button field within the dialogue of the size and position specified. The *depth* and *width* size of the field must be large enough for the radio button and its adjacent label which is written to the right of the button. The `label` is any atom. The radio button is a circle.

The *final_value* must be a variable. It will be bound to on or `off` indicating whether or not the button was in the selected state when the dialogue was read.

The *initial_value* is one of:

`on` The radio button is displayed initially selected with a dot in the circle.

`off` The radio button is displayed initially unselected. It is an empty circle.

NOTE The radio button formats of a dialogue format list must appear on the format list in groups of adjacent format items, each such adjacent group being separated from any other adjacent group of radio buttons by at least one non-radio button format descriptor. (The radio buttons of each adjacent group should also be displayed in the dialogue box as adjacent fields so that they are interpreted as a group of alternative buttons by the user.)

A radio button format *must not* be the last format descriptor of the format list. The group of radio button formats in the list must always be followed by at least one other different format item.

If you only want to have Ok and Cancel buttons (which must be the first two formats of the format list) and radio buttons in a dialogue, then follow the radio button formats with a dummy text field format which displays the empty string ''.

Unless the user uses a shift click, only one radio button of each adjacent group can be selected at a time. Normal clicking on a radio button selects that button and deselects any other selected radio buttons in the adjacent group of buttons. Shift clicking on an unselected radio button selects that button and leaves selected any other selected radio butttons. A selected radio button can be deselected without selecting any other radio button by shift clicking on the button.

6. `icon(down_pos,in_pos,depth,width,icon_id)`

This displays an icon of the size and position specified. The *icon_id* must be an integer identifying the icon.

Some icon ids that you can use, and the corresponding icons are:

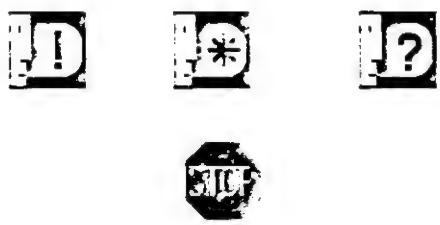
0 A rectangle containing a profiled speaking face on the left with a ! mark to the right.

1 A rectangle containing a profiled speaking face on the left with a * mark to the right.

2 A rectangle containing a profiled speaking face on the left with a ? mark to the right.

1003 A hexagon with STOP written in it.

(These are icons defined as Resources in the Macintosh System. Their appearance may vary according to the version of the System you have on your Mac.)



See the chapter on Advanced Dialogues if you wish to use your own icons.

DISABLING ITEMS

`disable(field_descriptor)`

Fields of type `button`, `check` or `radio` may be disabled by adding the modifier `disable` to their field format descriptor. The dialogue field is displayed 'greyed out' and cannot be selected or have its value changed by the user. This is useful for removing certain options on particular uses of the dialogue. For example,

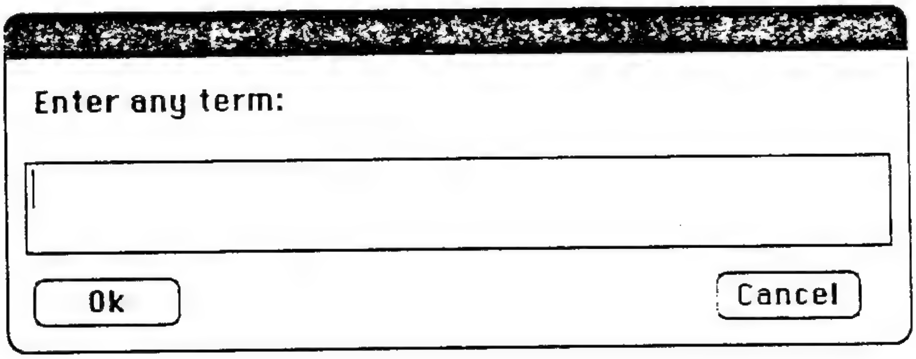
```
disable(button(50,50,20,90,'Continue'))
```

describes a disabled **Continue** button which would appear in grey in the dialogue.

EXAMPLES

1.
```
dialog('',50,60,120,370,
       [button(90, 10,  20,  60,'Ok'),
        button(90,290,  20,  60,'Cancel'),
          text(10, 10,  32,350,wseq(['Enter any term:'])),
          edit(45, 10,  32,350,'',read(Term))],
        Btn)
```

This is the call that implements the single argument call to the primitive `read`.



In this case the dialogue window has no title. It has just two buttons (items 1 and 2 on the format descriptor list), a text field, and an edit field into which the user will type a term.

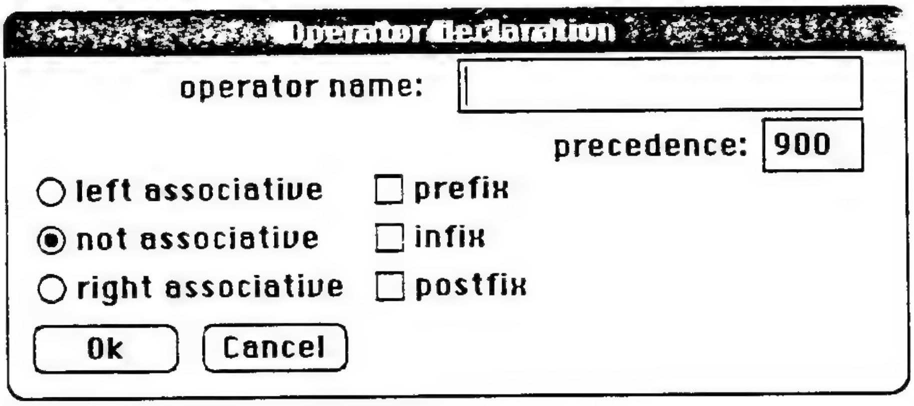
If the user clicks on the **Ok** button, `Btn` will have the value 1 and the the term typed into the edit field will be returned as the binding for the variable `Term`.

If the user clicks on the **Cancel** button then the `dialog` call will fail.

2.
```
dialog('Operator declaration',50,80,140,370,
      [button(110, 10, 20, 60,'Ok'),
       button(110, 80, 20, 60,'Cancel'),
         edit(  5,190, 16,160,'',Op),
        radio( 45, 10, 20,130,'left associative',off,Left),
        radio( 65, 10, 20,130,'not associative',on,Not),
        radio( 85, 10, 20,130,'right associative',off,Right),
         edit( 30,315, 16, 35,'900',read(Prec)),
        check( 45,150, 20, 70,prefix,off,Pref),
        check( 65,150, 20, 70,infix,off,Inf),
        check( 85,150, 20, 70,postfix,off,Post),
         text( 30,225, 16, 85,'precedence:'),
         text(  5, 70, 16,110,'operator name:')],
       Btn).
```

This is the call which displays the operator declaration dialogue in response to the **New operator...** menu command of the programming environment.

The variables `Op, Left, Not, Right, Prec, Pref, Inf, Post,` and `Btn` will all be bound by the call.



Notice that the **precedence:** edit field has its initial value supplied as the atom `'900'` whereas the **operator name:** edit field is initialised to the empty string. The 'labels' for these edit fields are actually separate text fields.

17.2 `mdialog` - display and read from an unlabelled modal dialogue

```
mdialog(down_pos,in_pos,depth,
        width,format_list,button)
```

ARGUMENTS

down_pos	: integer, position of top of dialogue as a number of pixels down from the top of the Mac display
in_pos	: integer, position of left of dialogue as a number of pixels in from the left of the Mac display
depth	: integer, depth of the dialogue in pixels
width	: integer, width of the dialogue in pixels
format_list	: list of dialogue field format descriptors
button	: variable, will be bound to an integer indicating which button was selected

USE

Almost exactly the same use as `dialog`. The difference is that the displayed dialogue box is completely rectangular and is unlabelled. It must also be responded to before anything else is done.

FORMAT DESCRIPTORS

All the format descriptors allowed in a `dialog` call can appear on the format list for an `mdialog` call. They have the same effect and there are the same constraints on their use. In addition, one other format descriptor is allowed. It describes a scrolling menu subfield:

```
menu(down_pos,in_pos,depth,width,item_list,
          preselected,selected)
```

This describes a scrolling menu subfield of the size and position specified. (Only *one* such menu may be specified for any dialogue.)

The *depth* of the menu must be the integer `16*n-1` where *n* is the number of atoms that will be on display in the scrolling menu at any time.

The extra arguments are:

item_list	: list of atoms
preselected	: list of atoms, sublist of *item_list*
selected	: variable, will be the list of user selected items from the menu

The atoms displayed in the scrolling menu will be the atoms in *item_list*.

The atoms in the *preselected* list will be shown as initially selected, i.e. will be in reverse video. (The *preselected* list may be empty.)

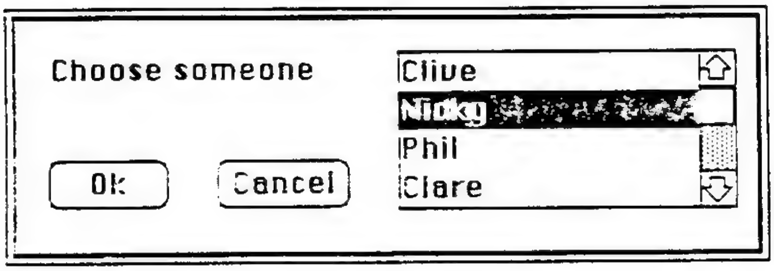
The user selects an item by clicking on it. By holding down the shift key whilst clicking, more than one item may be selected at a time. Items are deselected by clicking on them again.

When the user presses the **Ok** button the *selected* list will be the list of selected items.

EXAMPLE

The following call creates the modal dialogue shown below.

```
mdialog(60,100,90,300,
       [button(55, 10, 20, 50,'Ok'),
        button(55, 80, 20, 55,'Cancel'),
          text(10, 10, 20,145, wseq(['Choose someone'])),
          menu(10,155, 66,140,
       ['Clive', 'Nicky', 'Phil', 'Clare', 'Diane','Frank'],
       ['Nicky'],Selected)],
       Btn).
```



Notice that the depth of the menu is 66 which is 16*4-2, and thus 4 items are displayed in the menu field. The remaining two list items (`'Diane'` and `'Frank'`) can be displayed by clicking on the scroll bar.

The *preselected* list is the single item `'Nicky'` which appears initially in reverse video.

The user's eventual selection will be returned as the list `Selected`.

18 Advanced Dialogues

This chapter describes some of the more advanced features of MacPROLOG's dialogue handling facilities.

MacPROLOG supports the use of picture items in dialogues. Buttons and check boxes may be declared to be picture items. In the case of picture check boxes, there are also various 'selection modes' which can be specified, which determine the appearance of the item in its selected state (i.e. when the user has clicked on the item). The pictures can either be created using MacPROLOG Graphics, or they can come from resource files.

It is also possible to associate with any dialogue a goal, and for the dialogue to terminate only on successful completion of that goal. This can be useful in data entry applications, for example, where the dialogue in which the user enters data stays on the screen until it is correctly completed. In this case the dialogue's associated goal would perform checks on the entered information, give suitable messages, and, if necessary, modify the dialogue, until all the data was complete and correct.

There are primitives `setditem` and `moveditem` which change the appearance of a dialogue while it is on the screen - dialogue items may be repositioned, enabled or disabled, re-labelled or even removed altogether, without terminating the dialogue itself. The primitive `getditem` returns the current status of any dialogue item.

It is assumed that you have read the chapter on Dialogue Building, which describes how to set up a general modal or modeless dialogue.

18.1 Picture Items

In addition to the `edit`, `text`, `button`, `check`, `radio` and `icon` dialogue items described in the Dialogue Building chapter, there are the picture item types `pbutton` and `pcheck`.

Their field format descriptors are as follows.

18.1.1 **`pbutton(`*`down_pos, in_pos, depth, width, pict_descr`*`)`**

This creates a picture button field within the dialogue, of the size and position specified. (Remember that the *`down_pos`* and *`in_pos`* pixel values are relative to the dialogue window). The button is displayed, without a label, as the picture given by *`pict_descr`*.

The item behaves exactly like an ordinary button - when the user clicks on it the picture is displayed in reverse video, and its position in the format list is returned as the binding for the *`button`* variable in the `dialog` call.

Picture descriptions are fully explained in the chapters on Graphics. However, if you have pictures in resource files then *`pict_descr`* can be of the following form

```
        resource(name)
or      resource(name,file)
or      resource(name,file,vol)
```

where *`name`* is the picture name, *`file`* is an optional argument specifying the resource file, and *`vol`* an optional argument giving the file's volume id.

If *`file`* and *`vol`* are not given, the resource files that are currently open will be searched. If *`file`* is given, the file will be opened if necessary. If *`vol`* is not given, the current default volume is assumed.

18.1.2 pcheck(*down_pos*, *in_pos*, *depth*, *width*, *mode*, *initial_value*, *final_value*)

This creates a picture check box field within the dialogue, of the size and position specified. A picture is displayed instead of the usual check box and label, but the item behaves exactly like a normal check box. If the user clicks in the picture it becomes selected; clicking again will deselect it.

A picture description must be given to specify the appearance of the check box in its unselected state. There are various ways of displaying the check box in its selected state - the *mode* argument specifies the unselected picture `pict_descr` and the selection mode of the check box.

The *mode* argument must be one of:

`frame(pict_descr)`	The picture will be framed when it is selected.
`cross(pict_descr)`	The picture will have a 'check box cross' (as in an ordinary selected check box) drawn over the top when selected.
`outline(pict_descr)`	An attempt will be made to outline the picture's elements in bold (this only works well if the picture contains one or two simple, clearly defined objects) when it is selected.
`invert(pict_descr)`	The picture is 'inverted' on selection, i.e. displayed in reverse video.
`switch(pict_descr,pict_descr2)`	In this mode a second picture description must be given, and when the item is selected the second picture is displayed instead of the first. Clicking on this second picture deselects the item and the first picture will be displayed again.
`overlay(pict_descr,pict_descr2)`	In this mode a second picture description must be given, and when the item is selected the second picture is drawn on top of the first.

The `final_value` must be a variable. It will be bound to `on` or `off` indicating whether or not the picture check box was in the selected state when the dialogue was read.

The `initial_value` is one of:

`on`	The check box is displayed initially selected.
`off`	The check box is displayed initially unselected.

18.1.3 Icons

If you have your own icons, they may also be used in dialogues. The format descriptor is of the form

```
icon(down_pos,in_pos,depth,width,icon_descr)
```

where `icon_descr` takes the same form as a resource picture description, described above. (See the Graphics chapters for more details).

18.2 Extended Dialogues

A call to `dialog` or `mdialog` may be extended to include an extra term which specifies a goal to be associated with the dialogue. Whenever a button other than the **Cancel** button is clicked on, control is transferred to the goal specified. If the goal succeeds, the dialogue terminates. If the goal fails, the dialogue remains displayed and control returns to the user to complete or modify the dialogue entries.

The extended call takes the form

```
dialog(title,down_pos,in_pos,depth,width,
        format_list,button,goal)
```

where the arguments *title* to *button* are as described in the Dialogue Building chapter. The *format_list* may of course contain picture items. The call to `mdialog` can be similarly extended with an extra *goal* argument.

The *goal* argument must be a term of the form

test $(a_1, a_2, \ldots, a_k)$

where `test` is defined as a `k-2`-ary relation. This is because it will be called by the dialogue handler with two extra arguments which will be the dialogue ID and the number of the button pressed.

Whenever the user clicks on a dialogue button other than the **Cancel** button, a call is made to

test $(dlg, btn, a_1, a_2, \ldots, a_k)$

where *dlg* is the dialogue ID, and *btn* is the number of the button that was pressed. (These values may be used if you want to call `getditem`, `setditem` or `moveditem` within the `test` program - see below). If the call to `test` succeeds, the dialogue terminates. If the call fails, control returns to the dialogue - the dialogue remains on the screen and the user must take further action on the dialogue.

(Note that *goal* is only executed when the user clicks on a button other than the **Cancel** button. Actions such as clicking on radio items or check boxes, or entering text into an edit field, will not cause *goal* to be executed.)

When *goal* is called, all variables in the format descriptors will have been instantiated. For example if the dialogue contains a check box field, the *final_value* variable of its format descriptor will be either `off` or `on`, reflecting the current state of the check box. So the arguments $a_1, a_2, \ldots, a_k$ of *goal* can be output variables of the format descriptors, thus passing to the `test` program the current field values of the dialogue. However, the field values can also be accessed from within the `test` program using the `getditem` primitive.

18.3 `getditem` - get information on a dialogue item

`getditem(`*`dlg, item, type, abled, setting, value, rect`*`)`

ARGUMENTS

`dlg`	: integer, dialogue ID (inherited from call to `dialog` or `mdialog`)
`item`	: integer, the dialogue item number
`type`	: variable. Becomes the item's type
`abled`	: variable. Becomes either `enabled` or `disabled`
`setting`	: variable. Becomes either `on` or `off` or `null`
`value`	: variable. Becomes the item's value
`rect`	: variable. Becomes the item's enclosing rectangle

USE

This call is used to return the current status of any dialogue item. The item's parameters may be changed by calling `setditem` or `moveditem` - see below.

The *`dlg`* argument is the value passed by the dialogue handler (see discussion above).

The *`item`* number is the position of the dialogue item in the format list as specified in the original `dialog` or `mdialog` call (see the Dialogue Building chapter).

The *`type`* argument is instantiated to one of the following atoms corresponding to the item's type:

```
edit   text   button   pbutton   check   pcheck   radio   icon
```

The *`abled`* argument is instantiated to one of the atoms `enabled` or `disabled`, depending on the current status of the dialogue item.

The *`setting`* of the item is instantiated to one of the atoms `on` or `off` if the item is a button, check box or radio; or to the atom `null` if the item is of any other type.

The *`value`* argument returns the item's value according to the item's type, as follows:

`edit`	the textual contents of the edit field, as an atom
`text`	the item's text, as an atom
`button`, `check`, or `radio`	the item's label, as an atom
`pbutton`	the item's picture description
`pcheck`	either a single picture description or a list of two picture descriptions, depending on the *`mode`* specified in the `dialog` or `mdialog` call
`icon`	the atom `null`

The *`rect`* argument is instantiated to a term of the form

`box(`*`T, L, D, W`*`)`

where *T*, *L*, *D* and *W* are integers: *T* and *L* represent the position of the top left corner of the item as numbers of pixels from the top and left of the dialogue, and *D* and *W* give the item's pixel depth and width respectively.

18.4 **`setditem`** - set a dialogue item

`setditem(`*`dlg`*`, `*`item`*`, `*`reset`*`)`

ARGUMENTS

`dlg`	: integer, dialogue ID (inherited from `dialog` or `mdialog` call)
`item`	: integer, the dialogue item number
`reset`	: term representing the item's new status

USE

This call resets a dialogue item's status. The call may only be used to reset one parameter at a time; however, several calls to `setditem` may be made for any one dialogue item. To change an item's position, use `moveditem`.

The *`reset`* argument may take the following forms (some are appropriate only for certain types of dialogue items).

`enable`	The dialogue item is enabled.
`disable`	The dialogue item is disabled.
`on`	The dialogue item is set to on, i.e. it is selected
`off`	The dialogue item is set to `off`, i.e. it is deselected
`toggle`	The item is set to `on` if it was `off`, or to `off` if it was `on`.
`atom`	The item's text is changed to *`atom`* (e.g. may be the label of a button, or the contents of a text or edit field)
`picture`	The item's picture description is changed to *`picture`*.
`[`*`pic1`*`,`*`pic2`*`]`	The item's two pictures are changed to *`pic1`* and *`pic2`*.

NOTE The 'selection mode' of a picture dialogue item cannot be changed. Thus for example the `[`*`pic1`*`,`*`pic2`*`]` form of the *`reset`* argument is only appropriate for a picture item with a selection mode of either `switch` or `overlay`.

18.5 **`moveditem`** - reposition a dialogue item

`moveditem(`*`dlg`*`, `*`item`*`, `*`rect`*`)`

ARGUMENTS

`dlg`	: integer, the dialogue ID
`item`	: integer, the dialogue item number
`rect`	: term representing the item's new position

USE

This call repositions the specified *`item`* in the dialogue *`dlg`*.
The *`rect`* argument must be a term of the form

```
box(T, L, D, W)
```

where
T is the position of the top of the item as a number of pixels from the top of the dialogue
L is the position of the left of the item as a number of pixels from the left of the dialogue
D is the pixel depth of the item
W is the pixel width of the item

PRAGMATICS

A dialogue item may be made 'invisible' by specifying a new position which is outside the dialogue window. For example, if the dialogue window is 20[illegible] pixels deep and 300 pixels wide, the call

```
moveditem(Dial, Item, box(5[illegible], [illegible], 50, 5[illegible]))
```

will make the `Item` disappear. For instance, this may be done instead of disabling an item.

EXAMPLES

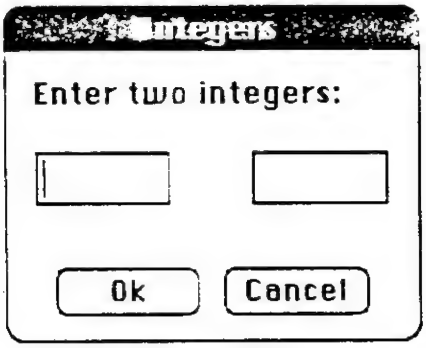
Two simple examples of the use of extended dialogues are given below.

Example 1

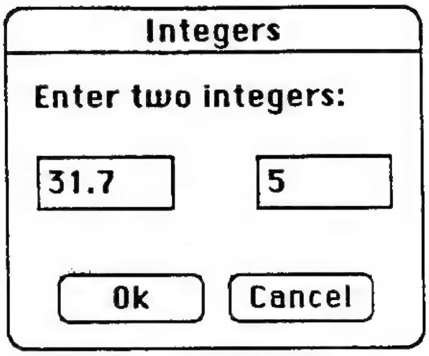
The first example displays a dialogue asking for two integers. The program `are_ints` simply checks that they are integers, and beeps and gives a warning message if they are not.

```
readints(I1,I2):-
        dialog('Integers',200,50,120,170,
        [button(90, 20, 20, 60,'Ok'),
         button(90, 90, 20, 60,'Cancel'),
           text(10, 10, 32,150,wseq(['Enter two integers:'])),
           edit(45, 15, 16, 50,'',read(I1)),
           edit(45,105, 16, 50,'',read(I2))],
        Btn, are_ints(I1,I2)).

are_ints(D,B,Int1,Int2):-
        integer(Int1),
        integer(Int2),!.

are_ints(D,B,Int1,Int2):-
        beep(10),
        message('You must enter integers'),
        fail.
```

The initial dialogue displayed is



If the user clicks on the **Ok** button and the two entries are not integers, the warning message is displayed. Note the explicit call to `fail` after the call to `message`, to ensure that control returns to the **Integers** dialogue window.



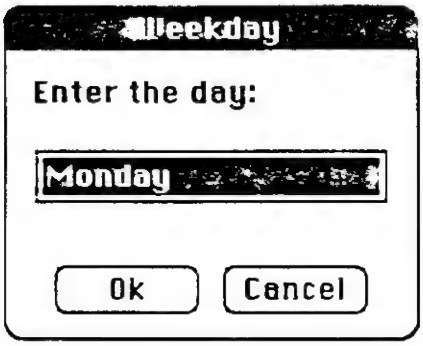
When the user has entered two integers, `are_ints` will succeed and the dialogue will terminate.

Example 2

In the second example `read_day`, the contents of the edit field is extracted in the checking program `is_day` by a call to `getditem`.

```
read_day(Day):-
        dialog('Weekday',200,60,120,1[illegible],
        [button(90,20,20, 60,'Ok'),
         button(90,90,20, 60,'Cancel'),
           text(10,10,32,150,wseq(['Enter the day:'])),
           edit(45,15,16,140,'Monday',Day)],
        Btn, is_day).

is_day(D,B):-
        getditem(D,4,Type,Abled,Set,Atom,Rect),
        on(Atom,['Monday','Tuesday','Wednesday',
                 'Thursday','Friday','Saturday','Sunday']),!.

is_day(D,B):-
        beep(10),
        message('That is not a day of the week!'),
        fail.
```

In this case the edit field is prefilled with the atom `'Monday'`.



If the user types anything not on the list of the days of the week and clicks on the **Ok** button, a warning dialogue will be displayed, as in the previous example.

(Of course whenever the user can only enter one item from a fixed list, as in this example, it is better to present the user with a scrolling menu of the items, rather than ask them to type into an edit field.)

19 Window Handling

In this chapter we document the various primitives for creating and manipulating windows. These primitives are used to implement most of the commands of the **Windows** menu of the programming environment.

There are extra primitives for manipulating graphics windows documented in the chapters on Graphics; however, many of the primitives in this chapter may be used for all types of MacPROLOG windows.

Windows may be of the following types

`prog`	for a program edit window
`disp`	for a display window
`info`	for an information window
`grap`	for a graphics window
`clip`	for the clipboard window

19.1 `wcreate` - create a display window

`wcreate(`*`name, visibility, down_pos, across_pos, depth, width`*`)`

ARGUMENTS

name	: atom, name of new window
visibility	: integer, either 0 or 1
down_pos	: integer, position of top of window as number of pixels down from the top of the Mac screen
across_pos	: integer, position of left of window as number of pixels in from the left of the Mac screen
depth	: integer, depth of window in pixels
width	: integer, width of window in pixels

USES

Creates a new display window (i.e. a window of type `disp`) with the given name and position. If the *visibility* is 0 then the new window is initially hidden. If it is 1, the window is visible.

You can read and write to this window using the window I/O primitives. No attempt is made to compile the contents of a display window, so there is no associated syntax or mode.

19.2 **`wpcreate` - create a new program edit window**

```
wpcreate(window_name,visibility,syntax,mode,
                down_pos,across_pos,depth,width)
```

ARGUMENTS

`window_name`	: atom, name of the window
`visibility`	: integer, either 0 or 1
`syntax`	: atom, syntax of window. Must be one of: `'<EDINBURGH>'` for Edinburgh syntax `'<USER1>'` for an optional user defined syntax `'<USER2>'` for another optional user defined syntax
`mode`	: atom, giving compilation mode. Must be one of: `'<COMPILE>'` for a Compiled window `'<INTERPRET>'` for an Interpreted window `'<DATA>'` for a Data window `'<OPTIMISE>'` for an Optimised window
`down_pos`	: integer, position of top of window as number of pixels down from the top of the Mac screen
`across_pos`	: integer, position of left of window as number of pixels in from the left of the Mac screen
`depth`	: integer, depth of window in pixels
`width`	: integer, width of window in pixels

USE

This call creates a new program edit window (a window of type `prog`) with the given name and position and the associated syntax and mode. (The **New...** command of the **Windows** menu in the programming environment calls `wpcreate`).

If the *`visibility`* is 0 then the new window is initially hidden. If it is 1, the window is visible.

The syntax specifier is the name of the compiler that will be used to compile the contents of the window when you use the **Check Program** command in the programming environment. (See also the description of the `windows` primitive below.)

<u>User Defined Syntaxes</u>

If you create a window with `<USER1>` or `<USER1>` as the associated compiler you must define this relation yourself. The compiler is called with two arguments. The first is the name of the window and the second is the atom which is the mode associated with the window. You should define it in the form:

```
'<USER1>'(Window,Mode):-
 cursor(Window,Front,Back),          /* find position of cursor */
 cursor(Window, 0, 0),               /* move cursor to beginning of window */
 compile_window(Window,Mode),        /*recursive read/compile of window clauses*/
 cursor(Window,Front,Back).          /* re-position cursor */
```

The `compile_window` program should use the window I/O primitives to read in the clauses of the window. It must then transform these into normal Edinburgh syntax clauses, perhaps using the `Mode` argument to select a special form of transformation, before adding them to the interpreted database using `assert`.

If you want to extend the programming environment so that new windows with syntax type `<USER1>` or `<USER2>` can be created by a menu command we suggest that you add a new menu using the `install_menu` primitive which has a command that displays a dialogue with an edit field to find the name of the new special syntax window, and perhaps a pair of buttons to select the `<USER1>` or `<USER2>` type (to which you can give more user friendly names). The command should then create the new window in some fixed position and of a fixed size using the appropriate `wpcreate` call.

(See the chapter on Implementing an Application for details of how this modification of the menu bar can be made automatic on the loading of an application program.)

19.3 **`wtype`** - get type of a window

`wtype(`*`name`*`, `*`type`*`)`

ARGUMENTS

name	: atom, name of a window
type	: atom or variable

USES

To find the type of a window, *`type`* must be a variable - it will then be bound to one of the following atoms:

`prog`	for a program edit window
`disp`	for a display window
`info`	for an information window
`grap`	for a graphics window
`clip`	for the clipboard window

If `wtype` is called with the *`type`* argument as one of the above atoms, the call succeeds if the named window is of this type, or fails otherwise.

19.4 **wsyntax** - find the associated syntax and mode of program window

wsyntax(*name,syntax,mode*)

ARGUMENTS

name	: atom, name of a window of type `prog`
syntax	: variable, will be bound to one of the atoms: `'<EDINBURGH>'` for Edinburgh syntax `'<USER1>'` for an optional user defined syntax `'<USER2>'` for another optional user defined syntax
mode	: variable, will be bound to one of the atoms: `'<COMPILE>'` for a Compiled window `'<INTERPRET>'` for an Interpreted window `'<DATA>'` for a Data window `'<OPTIMISE>'` for an Optimised window

USE

To find the associated syntax and mode of a named program window. The *syntax* and *mode* arguments must be variables. They will be bound to atoms giving the syntax and mode.

19.5 **wchg** - check if text of a program window has been changed

wchg(*name*)

ARGUMENT

name	: atom, name of window of type `prog`

USE

The call succeeds of the window has been edited since the last use of `wclchg` or since the window was loaded. It tests a flag which is set when the window is edited.

It is useful if you are implementing your own programming environment, and you want to define the equivalent of **Check program**. (See the description of `windows` below.)

19.6 **wclchg** - clear the edit flag for a window

wclchg(*name*)

ARGUMENT

name	: atom, name of window of type `prog`

USE

Clears the edit flag that is set when the text of a window is edited - the flag is tested using `wchg`.

19.7 **wrename** - rename a window

wrename(*old_name*, *new_name*)

ARGUMENTS

old_name	: atom, name of a window
new_name	: atom, new name of window

USE

To rename a window from *old_name* to *new_name*.

19.8 **wsearch** - search for text in a window

wsearch(*name*, *string*, *start*, *from*, *to*)

ARGUMENTS

name	: atom, name of a window
string	: atom, text to be found
start	: integer, character position for start of search
from	: variable, will become integer giving start position of found text
to	: variable, will become integer giving end position of found text

USE

To search for the occurrence of some sequence of characters in a window. The window should be of type `prog`, `disp` or `info`.

The sequence of characters are given as an atom in the *string* argument.

The *start* argument is the character position (relative to the start of the window) from which to commence the search. If this is given as 0, the search starts at the beginning of the window.

If the sequence of characters is not found, the call fails. If it is found, variable *from* becomes the character position of the start of the text and the *to* variable becomes the character position of the end of the text. These values can be then used in a call to the `cursor` primitive to highlight the found text.

19.9 `cursor` - find position of, re-position, or change appearance of cursor

```
cursor(window, from, to)
cursor(newcurs)
```

ARGUMENTS

window	: atom, name of a window
from	: variable or integer
to	: variable or integer
newcurs	: atom

USES

1. Three argument form
The three argument form of `cursor` is used to find or reposition the text cursor in a window. The *window* should be of type `prog`, `disp` or `info`.

To find the position of the cursor within a window the second and third arguments should be variables. *from* will then be bound to the character position of the start of the cursor and *to* will be bound to the character position of the end of the cursor. They will have different values only if the current cursor position is a selection range.

To re-position the cursor all arguments must be given. The cursor will become a selection range if *from* and *to* have different values.

A negative integer given as the value of *from* or *to* specifies the end of the text in the window. A value of 0 specifies the beginning of the text. The *to* value should be greater than (or equal to) the *from* value when they are both positive integers.

The call

```
cursor(window,0,-1)
```

will select the entire window (the cursor extends from the start to the end of the window's text).

2. One argument form
The single argument form is used to change the visual appearance of the mouse cursor. The cursor retains its new appearance until another call to `cursor` or it is changed by the underlying system (for example the cursor automatically beomes a watch during some system processes).

The *newcurs* argument must be one of the following names of predefined cursors.

`arrow`	the system's arrow pointer
`i_beam`	the text cursor
`spy_glass`	a 'magnifying glass'
`pen`	a pen
`cross_hair`	a cross
`thick_cross`	a large cross
`watch`	the system's wristwatch cursor
`garbage`	the MacPROLOG 'Hoover' garbage collector
`split`	a thick vertical line
`left_thumb`	a horizontal scroll bar's left hand pointer
`right_thumb`	a horizontal scroll bar's right hand pointer
`up_thumb`	a vertical scroll bar's upper hand pointer
`down_thumb`	a vertical scroll bar's lower hand pointer

19.10 **`wsltxt`** - select text from a given character range in a window

`wsltext(`*`window, from, to, text`*`)`

ARGUMENTS

`window`	: atom, name of a window
`from`	: integer, beginning of text
`to`	: integer, end of text
`text`	: variable, will become an atom

USE

To read in the sequence of characters in a given cursor range. The sequence of characters in the range *`from`* *`to`* are wrapped up as an atom which becomes the value of variable *`text`*.

PRAGMATICS

Can be used together with `cursor` and `wsearch` to implement a command that searches for the next occurrence of the currently selected text in a window, i.e. to implement the **Find selection** menu command of the programming environment.

19.11 **`cut`** - delete from the cursor range in a window

`cut(`*`window`*`)`

ARGUMENT

`window`	: atom, name of a window

USE

Text in the cursor selection range of *`window`* is deleted from the named window and placed in the clipboard. This corresponds to the **Cut** command of the **Edit** menu.

If *`window`* is a Graphics window, the selected pictures will be cut and placed in the clipboard using the currently set clipboard format for pictures. This format may be set using the **Defaults** command of the **File** menu (see the chapters on Graphics).

19.12 `copy` - copy from the cursor range in a window

`copy(window)`

ARGUMENT

`window` : atom, name of a window

USE

Text in the cursor selection range of *`window`* is copied from the named window and placed in the clipboard. If the cursor is an insertion point, this call does nothing. This corresponds to the **Copy** command of the **Edit** menu.

If *`window`* is a Graphics window, the selected pictures are copied to the clipboard, using the current clipboard picture format (see the chapters on Graphics).

19.13 `clear` - delete from the cursor range in a window

`clear(window)`

ARGUMENT

`window` : atom, name of a window

USE

Text in the cursor selection range of *`window`* is deleted from the named window. It is **not** placed in the clipboard. This corresponds to the **Clear** command of the **Edit** menu.

If *`window`* is a Graphics window, the selected pictures will be cleared.

19.14 `paste` - paste into the cursor range in a window

`paste(window)`

ARGUMENT

`window` : atom, name of a window

USE

Text in the clipboard is pasted into the cursor selection range of *`window`*, overwriting the text in the selection range. If the selection range is an insertion point, the clipboard text is simply inserted at that point. This corresponds to the **Paste** command of the **Edit** menu.

If *`window`* is a Graphics window and the clipboard contains a picture, the picture will be pasted into the window, and its picture description added to that window's list of pictures.

19.15 `undo` - undo the last editing action

`undo(`*`window`*`)`

ARGUMENT

`window` : atom, name of a window

USE

If *`window`* is the window in which the most recent text editing was done, the last editing action is undone. This corresponds to the **Undo** command of the **Edit** menu.

19.16 `whide` - hide a window

`whide(`*`window`*`)`

ARGUMENT

`window` : atom, name of a window

USE

The named window is hidden. It can be made visible again using the `wshow` primitive.

Note that the contents of the window are preserved and you can still read and write to the window using the window I/O primitives.

19.17 `wshow` - make visible a hidden window

`wshow(`*`window`*`)`

ARGUMENT

`window` : atom, name of a window

USE

The named window is made visible if it was hidden. Otherwise, there is no change. *`Window`* does not automatically become the front window - you must use `wfront` to achieve this.

19.18 `wfront` - make a window the front window

`wfront(window)`

ARGUMENT

`window` : atom or variable, name of a window

USE

If *`window`* is given, the named window is made the front window even if it was hidden.

If *`window`* is a variable, the name of the current front window is returned as the value of *`window`*.

(If the front window is the **Default Output Window**, the name returned will be `'Default Output Window'`, and not `user`.)

19.19 `wvis` - test if a window is visible

`wvis(window)`

ARGUMENT

`window` : atom, name of a window

USE

The call succeeds if the named window is currently visible, and fails if it is currently hidden.

19.20 `wkill` - kill a window

`wkill(window)`

ARGUMENT

`window` : atom, name of a window

USE

The named window is deleted. It cannot be re-displayed using `wshow`. The text of the window is lost, but the compiled relation definitions or interpreted clauses produced by compiling the window are *not* deleted.

If *`window`* is a Graphics window all picture descriptions associated with the window are removed.

If the *`window`* argument is not the name of a current window then `wkill` does nothing and succeeds.

19.21 **`wsize`** - get or set the size and position of a window

`wsize(`*`window, down, in, depth, width`*`)`

ARGUMENTS

`window`	: atom, name of a window
`down`	: integer or variable, the position of top of *`window`* as number of pixels down from top of the Mac display
`in`	: integer or variable, the position of left of *`window`* as number of pixels in from left of the Mac display
`depth`	: integer or variable, pixel depth of *`window`*
`width`	: integer or variable, pixel width of *`window`*

USE

To find or set the position and size of a window.

The *`window`* argument must be an atom, the name of a window.

Any or all of the remaining arguments may be variables: if so, they will be instantiated to the current values for the named window.

Any or all of the remaining arguments may be integers: in this case the corresponding attribute of the named window will be set to the given value.

For example, the call

```
wsize('My Window', Top, Left, 200, 300)
```

will set the size of the window `'My Window'` to be 200 pixels deep and 300 pixels wide. The position of the top left corner of the window (which is unaffected by this call) is returned as the values of `Top` and `Left`.

The call

```
wsize('My Window', 50, 70, Depth, Width)
```

will position the window `'My Window'` 50 pixels from the top and 70 pixels from the left of the screen. The current size of the window (which is not affected by this call) is returned as the values of `Depth` and `Width`.

19.22 `windows` - find all windows of a given type

`windows(type, window_list)`

ARGUMENTS

`type`	: atom, either `prog`, `disp`, `info` or `grap`
`window_list`	: variable, will become list of window names of specified type

USE

Finds all the names of the windows, visible or hidden, of the given *`type`*.

The primitive can be used to implement the equivalent of **Check program** as follows:

```
'Check program':-
        windows(prog,Wlist),
        on(Window,Wlist),                  /* find a program window */
        wchg(Window),                      /* has window been edited? */
        wsyntax(Window,Syntax,Mode),       /* get syntax and mode */
        wclchg(Window),                    /* reset its edit flag */
        Syntax(Window,Mode),               /* call Syntax compiler */
        fail.                              /* backtrack for next window */
'Check program'.
```

19.23 `wfont` - font details for a window

`wfont(window, font, face, size)`

ARGUMENTS

`window`	: atom, name of window
`font`	: atom or variable
`face`	: integer or variable
`size`	: integer or variable

USE

To set or find the font details of a window.

The *`font`* is an atom representing the font name, such as `'Chicago'`, `'Times'`, `'Courier'` or any other font supported by your system disk.

The *`face`* is an integer representing the style of the text. It may be

0	Normal
1	**Bold**
2	*Italic*
4	Underline
8	Outline
16	Shadow

or the sum of any of these (to represent a combination of styles).

The *`size`* represents the font size, and may be 10, 12, 14, 18, etc.

If the *`font`*, *`face`* and *`size`* arguments are variables, they will be bound to the current details for the named *`window`*.

19.24 `balance` - check balance of brackets in text

`balance(window, from, to)`

ARGUMENTS

window	: atom, name of a window of type `prog` or `disp`
from	: variable
to	: variable

USE

The current selection range in the named window is extended to include the next outer pair of matching brackets. The recognised brackets are

( ) { } [ ]

The variables *from* and *to* are given the values of this new cursor selection range.

This is the same as the **Balance** menu command of the programming environment.

19.25 `screen` - find size of the Mac display

`screen(depth, width)`

ARGUMENTS

depth	: variable, will become depth of your Mac display in pixels
width	: variable, will become width of your Mac display in pixels

USE

To find the size of the screen of the Macintosh on which a program is running. On a Macintosh Plus the *depth* will be 342 and the *width* will be 512.

19.26 `cleanup` - tidy up the screen window display

```
cleanup(type,top,left,top_incr,left_incr)
cleanup(type,top,left,top_incr,left_incr,
                depth,width)
cleanup(type,top,left,top_incr,left_incr,
                depth,width,font,size)
```

ARGUMENTS

type	: atom
top	: integer
left	: integer
top_incr	: integer
left_incr	: integer
depth	: integer
width	: integer
font	: atom, font name
size	: integer

USE

To rearrange on the screen all the windows of a given type.

1. Five argument call
The *type* argument is the window type, which should be one of

```
prog
info
disp
grap
```

The *top* and *left* arguments give the distance, in pixels, from the top and left of the screen respectively that the first window should be placed.

The *top_incr* and *left_incr* arguments give the distance between the top left hand corners of successive windows.

For example if *top_incr* has the value 5 and *left_incr* has the value 8 then the top left hand corner of each window will be 5 pixels down and 8 pixels across from the previous one.

2. Seven argument call
The first five arguments are as above. In addition, each window may be resized as it is replaced on the screen. The *depth* and *width* arguments give the new size (in pixels) for the windows.

3. Nine argument call
If the last two arguments are given, each window will have its text converted to the font *font* of size *size*.

20 Resources

MacPROLOG provides access to the Resource Management routines of the Macintosh. You may create resource files (or create a resource fork for an existing data file), and load and save resources from these and other existing resource files. The MacPROLOG Graphics language and Graphics primitives provide for extensive use of picture, icon and cursor resources. In addition, you may write code resources in either C or Pascal and call them directly from within Prolog.

Further details on specific uses of the various resource types are given in the chapters on Graphics, Dialogues and the Pascal and C Interface chapters.

20.1 Resource Items

QuickDraw pictures, cursors, icons and other objects used by the Macintosh can exist on disk as resource items in special resource files.

Resource items have the following attributes.

a) resource type
There are many resource types. Those used in MacPROLOG are

`'PICT'`	a QuickDraw picture
`'ICON'`	an icon
`'CURS'`	a cursor

b) resource item number
This is a number unique to that file, which can always be used to identify and access the resource item within the file.

c) resource item name
This is the name for the resource item in the file. This may be the empty string '', especially for resource items created outside of MacPROLOG.

d) resource file name
This is the name of the resource file. Resources can be associated with existing program files, as these files can have two 'forks': a *data* fork (where your program is stored) and a *resource* fork for your resources.

e) resource file volume
The volume ID where the resource file resides - this can be found using the `old` primitive.

20.2 Using Resource Items in MacPROLOG™

A resource item cannot be loaded until its resource file has been opened. The MacPROLOG™ application resource file will always be open, as will the System resource file.

Generally, when resource items are first accessed in MacPROLOG the named resource file in which they are stored is automatically opened. Where a primitive has optional arguments for a resource file name, you only need to supply this information if the file is not (or may not be) already open. However, you may sometimes need to open resource files explicitly with the `res_open` primitive, described below.

When a resource file is first opened, a copy of its 'memory map' i.e. a list of the resources that are in that file, is stored in memory. When memory is tight, these memory maps can be removed using `res_close`.

You can get a list of all resource items of a given type in currently open resource files by calling `res_items`.

When a resource is to be loaded, the Macintosh Resource Manager searches through all the open resource files, searching the most recently opened files first. Once a resource item is in memory, any subsequent uses of it are very fast. You can delete a resource item from memory if you have finished with it and are running short of space, using `res_finish`. Then if you need the item again it will be reloaded from disk.

Individual resource items may be loaded and used by calling specific primitives. The following are some of the primitives you may use; they are fully documented elsewhere.

Resource **pictures** can be accessed using the Graphics Description Language (GDL) `picture` descriptor and the `pbutton` or `pcheck` dialogue format descriptors. They can be created and saved to disk using `save_pic`.

Resource **icons** can be accessed using the GDL `icon` descriptor and the `icon` dialogue format descriptor.

Resource **cursors** can be accessed using `gcursor`.

Code resources may be loaded and run using the `call_c` and `call_pascal` primitives.

20.3 **res_create** - create a resource file

res_~~open~~ create (*file*)
res_~~open~~ create (*file, volume*)

ARGUMENTS

file	: atom, name of resource file
volume	: integer, volume identifier

USE

To create a named resource file.

If *file* does not already exist, it will be created, initially with an empty resource fork and an empty data fork.

If *file* already exists but has no resource fork, a resource fork will be created for the file.

If *file* already exists and has a resource fork, then `res_create` does nothing.

`res_create` does not open the file - to do this you must call `res_open`.

PRAGMATICS

A MacPROLOG program file initially has a data fork (where the program is stored) and no resources. Thus if you call `res_create` with an existing MacPROLOG program file name, you can create a resource fork for that file. You can then store resources (such as pictures, icons etc.) with the program. Because the data fork and the resource fork of a file are logically separate, you can save a new version of the program to the file without affecting any resources you have already saved.

20.4 `res_open` - open a resource file

```
res_open(file)
res_open(file, volume)
```

ARGUMENTS

file	: atom, name of resource file
volume	: integer, volume identifier

USE

To open a named resource file, thereby making all its resources accessible.
(You don't ever need to open the System or the MacPROLOG resource file, since they always remain open).

20.5 `res_close` - close a resource file

```
res_close(file)
```

ARGUMENTS

file	: atom, name of resource file

USE

To close a named resource file. Its resources are no longer directly accessible.
If *file* is not the name of a currently open resource file then `res_close` does nothing and the call succeeds.

WARNING: Do not ever attempt to close the System or the MacPROLOG resource file, since they need to be open for access all the time.

20.6 `res_finish` - delete a resource item from memory

```
res_finish(nameornum)
```

ARGUMENTS

nameornum	: atom or integer, name or number of resource item

USE

To remove a resource item from memory. Use this if you are running short of memory, and the resource item (say, a picture) is large. Note that this will not close the resource file in which the item is stored, and you may still use the item, but any subsequent reference to the resource will generate a disk access to reload the resource item.

20.7 `res_items` - return information on available resource items

`res_items(`*`res_type,listofitems`*`)`

ARGUMENTS

res_type	: atom, resource type
listofitems	: variable, will be bound to list of resources

USE

This primitive returns information on all the resources of a given type that can be found in all the currently opened resource files.

The *res_type* argument is a four character atom representing the resource type, for example 'PICT', or 'ICON'.

The *listofitems* is a list of which each item is a three element list of the form

[ResourceName, ResourceNum, File]

ResourceName is an atom that labels the resource item. If for example it is a picture saved in MacPROLOG by a call to `save_pic` it will be the name used in that call. Resources created in other applications often have the empty string ('') as their name: they tend to rely on the resource number to identify the resource within a file.

ResourceNum is a unique integer identifying the resource item within that file.

File is the name of the resource file. For resource items residing in the System or in MacPROLOG, this file name will be `system`.

21 The C Interface

21.1 Introduction

MacPROLOG™ allows you to write your own primitives in C and call them directly from within Prolog. The primitives must be compiled as separate code resources. and they are then loaded as they are needed by the Macintosh Resource Manager. This means that there is no need to rebuild the LPA MacPROLOG™ application file, but you can create your own separate library of C routines which are callable from Prolog. The extent to which these new primitives behave like a program written in pure Prolog depends on how many of the primitive's uses have been catered for by the C programmer.

A C primitive may have arguments passed to it from Prolog. There is a collection of interface routines which allow you to access these arguments and their values, and return values to Prolog by binding the variables of the call.

Like a Prolog program, a C primitive either succeeds or fails, possibly binding variables in the process. Unlike a Prolog program, however, the C primitive cannot be non-deterministic: that is, you cannot backtrack into it to evaluate alternative solutions. If you want this behaviour, you can write a Prolog program to sequence through non-deterministic solutions to your C primitive.

Having written your new primitive as a code resource, it is called from within Prolog with the `call_c` primitive, which will load and run the specified code. Prolog arguments to the call may be accessed from within the C code.

21.1.1 Requirements

To use the C Interface, you will need the supplied library of Prolog interface functions, a C Compiler and Linker, and a 68000 Assembler. In particular, your C compiler must be able to compile functions which conform to Pascal calling conventions, since the interface functions are declared as "Pascal" functions (which corresponds to the MacIntosh Toolbox calling conventions).

Important Note
LPA MacPROLOG™ was written using the MPW (Macintosh Programmers' Workshop) package, and instructions for creating code resources here refer to MPW. If you use a different compiler, there may be differences in function calling conventions, and you may have to make some adjustments to the supplied source files or to the way in which you call the Prolog interface functions. See the Technical Specification Section for further details.

21.2 Calling a C Function - An Outline

Here we sketch the format of a C primitive and the method of calling it from Prolog. For a more detailed description, refer to later sections of this chapter. In particular, a complete example is given at the end of this chapter.

There is a file `cxface.hed` which contains the various function declarations needed to interface with Prolog, which should be included in your C source file. There is a compiled assembly language file `xface.o` which contains the function definitions, which should be linked with your compiled C code.

When a C program is called from Prolog it will be called with two arguments.

The first argument (`argc` in the example below) is an integer giving the number of Prolog arguments to the call. (You can read these Prolog arguments using the `get_arg` function described later in the Term Access Functions section.)

The second argument is a pointer to a routine in the MacPROLOG™ application code which the various Prolog term access routines use. This second argument must be passed on to the `set_link` function (defined in the interface files) as the first function call of your C code; thereafter this `link` argument may be ignored. The `set_link` function does the "gluing" of your code resource to MacPROLOG™.

The C primitive must return a Boolean value (i.e. 0 or 1) corresponding to the primitive succeeding or failing. The interface file defines the type `bool`, and also defines `TRUE` and `SUCCESS` to be equal to `1`, and `FAIL` and `FALSE` to be equal to `0`.

Your C program will look something like this.

```
bool mycprim(argc,link)
 int argc;
 void (*link)();

 {
  ... variable declarations etc.

  set_link(link);
  if (argc != 3)
   return(FAIL);

  ... rest of program

  return(SUCCESS);
 }
```

As described above, `argc` is the Prolog argument count. This particular program is expecting 3 Prolog arguments, and it immediately fails if this requirement is not met. The body of the program will make calls to the Prolog term access routines to read the three Prolog arguments, and to the term construction functions which will bind any variables of the call.

The C code must then be compiled and linked as a code resource (you may give it any resource type and ID). It is then called from MacPROLOG using the `call_c` primitive (see below).

21.3 `call_c` - load and run a C code primitive.

`call_c(`*A_1*`,`*A_2*`, . . . ,`*A_k*`, `*`restype`*`, `*`resid`*`)`

ARGUMENTS

A_1,A_2, ...,A_k	: Prolog arguments to the C primitive
restype	: resource type of code resource
resid	: resource ID of code resource

This call loads and runs a C routine in the code resource identified by the *`restype`* and *`resid`* arguments. The first *k* arguments to `call_c` will be placed in the Prolog argument registers for subsequent access by the C primitive.

Note that the resource file containing the resource will *not* be automatically opened by `call_c`: you should do this with a call to `res_open` before you call the code.

The `call_c` primitive will succeed or fail depending on the `SUCCESS` / `FAIL` result passed back from the C program. `call_c` will also fail if the resource cannot be loaded for any reason. This would happen if, for example, the resource file was not open.

EXAMPLE

Suppose you have a C program in a code resource of type `MINE` and with ID `100`, which expects 3 arguments. If in Prolog you name this primitive `cprim`, then you would add the following clause.

```
cprim(A,B,C) :-
  call_c(A,B,C,'MINE',100).
```

After you have opened the resource file containing your code resource (by calling `res_open`), `cprim` can be called just like any other Prolog program.

21.4 Storage of Terms in MacPROLOG

In order to make use of the functions providing an interface to Prolog from your C program, you need to be aware of how terms are represented internally in MacPROLOG. You do not have to understand the *exact* memory formats, but you need to know the general structure of the different sorts of Prolog term and how they are stored.

21.4.1 Tagged Cells

A term in MacPROLOG is stored in a *tagged cell*. The *tag* of the cell indicates the type of data stored in the cell, and the rest of the cell represents its value.

Access to these cells from C is always via *cell pointers*. The type `cellpo` is defined in the C interface to be a tagged cell pointer.

All term manipulation is done by passing cell *pointers* to the various term access and construction functions provided. You will never manipulate the tagged cells directly.

The C interface provides a number of these term access functions which allow a C program to find out about the term stored in a particular cell. For example, the `get_int_val` function returns the integer value of an integer cell. There are also a number of term construction functions which allow new Prolog terms to be built and returned as bindings for existing unbound variables. For example, the `put_list` function binds a variable to a Prolog list structure. These functions are described in more detail later.

These term access and construction functions present a fairly abstract view of the MacPROLOG data structures, and therefore you need not be over concerned with the exact internal representation and memory layout of Prolog terms.

21.4.2 Prolog Argument Registers

A C function can access the Prolog arguments given in the call to `call_c`. There are 31 Prolog argument registers each of which can hold any Prolog term (i.e. a tagged cell). There may therefore be up to 31 Prolog arguments to the C function which will be placed in these argument registers. The interface function `get_arg` can then be used from within C to read these arguments. `get_arg` will return a pointer to the term cell in a specified Prolog argument register (see the Term Access Functions section below).

21.5 Prolog Term Structures

There are seven different term types in LPA MacPROLOG™. These are detailed below. As described above, each Prolog term is held in a tagged cell. The tag values (defined in the interface files) are as follows.

TERM TYPE	TAG
Variable	VARTAG
Integer	INTTAG
Real	REALTAG
Constant	CONTAG or STRTAG
List	LISTTAG
Empty List	NILTAG
Tuple	TPLTAG

21.5.1 Variable

A `VARTAG` cell represents an unbound variable. Such cells may be bound to other Prolog terms using the term construction primitives such as `put_int_val`, `put_con_text` etc.

21.5.2 Integer

An `INTTAG` cell represents an integer, stored as a 24 bit signed value. Note that C uses 32 bits for integers, but integers outside the range -8388608 to 8388607 will not be correctly represented in Prolog.

21.5.3 Real

A `REALTAG` cell represents a floating point number. It is stored as an 80 bit value, which corresponds to the C data type `extended`.
(Note that in MacPROLOG™ real numbers are automatically converted to integers if possible, so that when you construct a real cell you may sometimes get an integer cell instead).

21.5.4 Constant

A `CONTAG` or `STRTAG` cell represents some text stored as a sequence of ASCII characters. The cell may have either of the two tags `CONTAG` or `STRTAG`, which merely reflects whether the text is stored as a 'permanent' dictionary entry or as a 'temporary' object in the Prolog 'heap'. The actual format of the stored text is identical in each case.

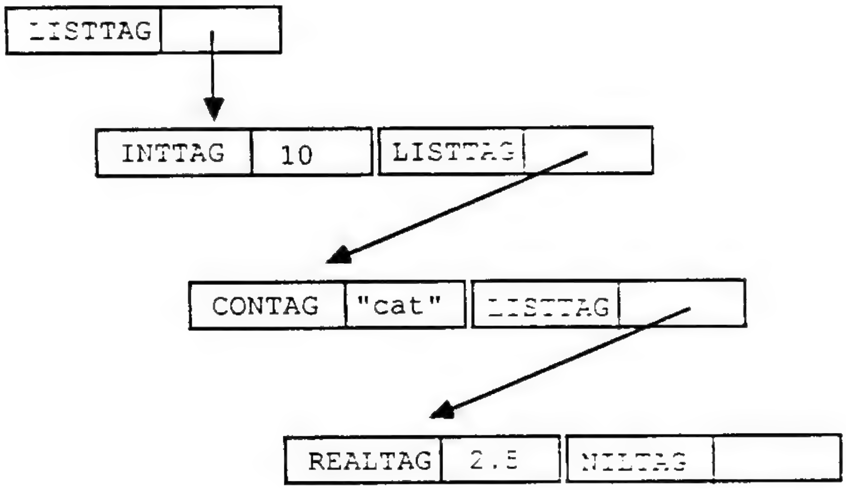
21.5.5 List

A `LISTTAG` cell represents a Prolog list structure. The list cell actually points to two more term cells, one representing the term at the head of the list, and the other representing the tail. The tail will usually be another `LISTTAG` cell, or the empty list (i.e. a `NILTAG` cell).

For example, the Prolog list

```
[10, cat, 2.5]
```

is represented by a `LISTTAG` cell which points to an integer cell with the value `10`, and another `LISTTAG` cell. This second list cell points to a constant cell with the value `cat`, and a third list cell, which in turn points to a real cell with value `2.5`, and a `NILTAG` cell.



Thus the internal representation of the above list is

```
[10, [cat, [2.5, []] ] ]
```

21.5.6 Empty List

A `NILTAG` cell represents the empty list. As described above, one of its uses is as a 'tail cell' to terminate a list. A `NILTAG` cell has no "value".

21.5.7 Tuple

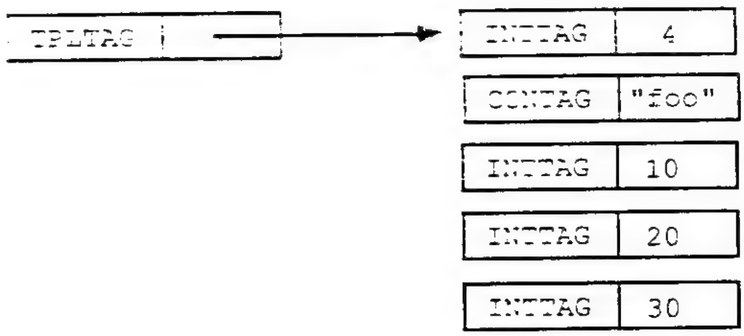
A TPLTAG cell represents a Prolog tuple structure. A tuple is a fixed size array of elements, where each element is any Prolog term. This corresponds to the internal representation of a functor and its arguments.

For example, the Prolog term

```
foo(10,20,30)
```

is represented internally as a tuple with four elements, the first element being a constant cell with value foo, and the remaining elements being integer cells with values 10, 20 and 30. (MacPROLOG also stores an extra "size" cell at the beginning of the tuple structure which gives the number of tuple elements.)

The above tuple therefore looks like this.



Since a tuple is a fixed size, it has a more compact representation than a list.

21.6 C Interface Term Access Functions

These functions allow you to read Prolog terms from within C. As mentioned above, a Prolog term is stored in a tagged cell, but all term manipulation is done using *pointers* to these cells.

All these functions are declared in the file `cxface.hed` (which you will `#include` in your C source file).

Note that these functions will do *no* checking of arguments: you are responsible for ensuring that any arguments you pass to these routines are of the correct type. For example, if you pass a `LISTTAG` cell pointer to the function `get_con_text`, there is no guarantee of what the results might be!

The types `cellpo` (a tagged cell pointer) and `real` (an 80 bit floating point number) are declared in the file `cxface.hed`.

21.6.1 get_arg

```
cellpo get_arg(num)
short num;
```

This function returns a pointer to the cell in Prolog argument register `num`, where `num` ranges from `1` to `argc` (the Prolog argument count, passed as a parameter to your C function). You may read the Prolog argument registers in any order.
(There is no guarantee that you will get anything sensible if you try to read a register beyond `argc`. The validity of `num` is not checked by `get_arg`.)

21.6.2 get_tag

```
short get_tag(cp)
cellpo cp;
```

This function returns the tag of a Prolog cell pointed to by `cp`. The value returned will be one of:

`VARTAG INTTAG REALTAG CONTAG STRTAG LISTTAG NILTAG TPLTAG`

The tag indicates the type of the data stored in the cell.
For example,

```
get_tag(get_arg(1))
```

will return an integer representing the type of the first Prolog argument.

21.6.3 get_int_val

```
int get_int_val(cp)
cellpo cp;
```

This function returns the integer value of an `INTTAG` cell pointed to by `cp`. As described above, although a C `int` type is returned, only the lower 24 bits are used by MacPROLOG. (The integer returned by `get_int_val` will have been sign extended to 32 bits).

21.6.4 `get_real_val`

```
real get_real_val(cp)
cellpo cp;
```

This function returns the real (floating point) value of a `REALTAG` cell pointed to by `cp`. The type `real` is an 80 bit floating point number and is defined in `cxface.hed`.

21.6.5 `get_con_text`

```
char *get_con_text(str,cp)
char *str;
cellpo cp;
```

This function returns the text of either a `CONTAG` or `STRTAG` cell pointed to by `cp`. The text will be copied as a null terminated string into the buffer pointed to by `str`; this pointer is also returned as the function result.
You *must* ensure that `str` points to a buffer large enough to hold the string; the maximum length of a MacPROLOG™ constant is 255 characters. The `get_con_text` function does *not* allocate the space for you.

21.6.6 `get_con_len`

```
int get_con_len(cp)
cellpo cp;
```

This function returns the actual number of characters in the text of the `CONTAG` or `STRTAG` cell pointed to by `cp`.

21.6.7 `get_list_head`

```
cellpo get_list_head(cp)
cellpo cp;
```

If `cp` points to a `LISTTAG` cell, this function returns a pointer to the cell representing the term at the head of the list.

21.6.8 `get_list_tail`

```
cellpo get_list_tail(cp)
cellpo cp;
```

If `cp` points to a `LISTTAG` cell, this function returns a pointer to the cell representing the tail of the list.

21.6.9 `get_list_len`

```
int get_list_len(cp)
cellpo cp;
```

If `cp` points to a `LISTTAG` cell, this function returns the length of the list (i.e. the number of elements in the list).

21.6.10 `get_tpl_len`

```
int get_tpl_len(cp)
cellpo cp;
```

If `cp` points to a `TPLTAG` cell, this function returns the number of elements in the tuple.

21.6.11 `get_tpl_nth`

```
cellpo get_tpl_nth(nth,cp)
int nth;
cellpo cp;
```

If `cp` points to a `TPLTAG` cell, this function returns a pointer to the cell representing the `nth` element of the tuple. The first element of the tuple is numbered 1. The value of `nth` should not exceed the length of the tuple.

21.7 C Interface Term Construction Functions

The following functions allow you to construct new Prolog terms and to bind existing unbound variables to them. Each function expects to be passed a pointer to an unbound variable (i.e. a pointer to a VARTAG cell), which will then be instantiated to a new Prolog term. The function returns a pointer to this newly created term.

In a similar way to the term access routines, no check is made that the cell pointer does in fact point to a VARTAG cell.

These bindings will automatically be undone on backtracking.

You should not try to construct a new term in a cell which is not a variable, as this is not logical and may lead to strange results.

21.7.1 put_int_val

```
cellpo put_int_val(unb,val)
cellpo unb;
int val;
```

This function constructs an integer cell and binds it to the variable pointed to by unb. As mentioned previously, only the lower 24 bits of val are stored.

21.7.2 put_real_val

```
cellpo put_real_val(unb,val)
cellpo unb;
real val;
```

This function constructs a real cell and binds it to the variable pointed to by unb. As noted previously, if val can be expressed as an integer, an integer cell will be constructed instead. Thus the tag of the constructed cell may be either REALTAG or INTTAG.

21.7.3 put_con_text

```
cellpo put_con_text(unb,str)
cellpo unb;
char *str;
```

This function constructs a constant cell from the null terminated string pointed to by str and binds it to the variable cell pointed to by unb. The tag of the returned cell will be either CONTAG or STRTAG.

21.7.4 put_list

```
cellpo put_list(unb)
cellpo unb;
```

This function constructs a list cell and binds it to the variable pointed to by unb. Two extra cells are created, one for the head of the list and one for the tail. Both of these will be initialised to unbound variables. You can access these cells (and bind new Prolog terms to them) using the get_list_head and get_list_tail functions.

21.7.5 put_nil

```
cellpo put_nil(unb)
cellpo unb;
```

This function constructs an empty list cell (a NILTAG cell) and binds it to the variable pointed to by unb.

21.7.6 put_tpl

```
cellpo put_tpl(unb,size)
cellpo unb;
int size;
```

This function constructs a tuple cell and binds it to the variable pointed to by unb. An array of extra cells is created for the size tuple elements. All of these will be initialised to unbound variables. You can access these cells (and bind new Prolog terms to them) using the get_tpl_nth function.

21.7.7 put_copy_cell

```
cellpo put_copy_cell(dest,src)
cellpo dest,src;
```

This function copies the term pointed to by src to the term pointed to by dest. The function returns a pointer to the dest term.
The destination cell must be an unbound variable at the time of the call, but the source cell may be *any* Prolog term (including another unbound variable). If you use this function to copy one variable cell to another, the effect is to make both cells refer to the same variable, and binding one of these variables to a new Prolog term automatically binds the other variable to the same term.

21.8 Reporting Errors from a C Primitive

If you wish to report an error from within your C code, you may call the function `errortrap` with an error code signalling the type of the error.

21.8.1 `errortrap`

```
void errortrap(errno)
short errno;
```

This function generates a Prolog numbered error. It does *not* return control to the calling routine, but jumps straight to the Prolog error handler. The error will be handled in exactly the same way as any other Prolog error. See the chapter on Implementing an Application for more details on error handling, and the Appendix for details of the error numbers available.

21.9 Creating a C Primitive

NOTE This documentation assumes that you are using the MPW C Compiler, 68000 Assembler, and Linker. If you are using a different Compiler and Linker you may need to refer to your own manuals for specific instructions on creating code resources.
See the Technical Specification section to determine what changes, if any, you may need to make to the C or Assembly language source code provided.

To create a new C primitive for MacPROLOG you will need the files `xface.o`, `cxface.hed` and `user.c`.

`xface.o` is an object code file containing the routines giving access to the Prolog data structures. It is written in assembly language, and needs to be linked in with your C object code to create the final code resource.
Note The source for `xface.o` is in the file `xface.a`, which you will only need to use if you find that the format of `xface.o` is not compatible with your Linker. In this case, you will need to assemble `xface.a` using an appropriate 68000 assembler (after making any necessary textual changes to conform to your assembler's syntax and other features), before linking the resulting object code with your C object code file.

`cxface.hed` is a text file to be included in your C source code file, `user.c`. It contains all the type and function declarations needed to interface with Prolog.

`user.c` is a text file which will contain the source of your C primitive.

21.9.1 The Format of a C primitive

When a C program is called from Prolog it will be called with two arguments.

The first is an integer giving the number of Prolog arguments to the call. (You can read these Prolog arguments using the `get_arg` function, described above in the Term Access Functions section.)

The second argument is the pointer to a routine in the MacPROLOG™ application code which the various Prolog term access routines use. All you need to do with this second argument is pass it on to the `set_link` function as the first function call of your C code: thereafter you should ignore this "link" argument. The `set_link` function does the "gluing" of your code to MacPROLOG™: if you omit to call `set_link` you can expect to have serious errors!

The C primitive must return a Boolean value (i.e. 0 or 1) corresponding to the primitive succeeding or failing. The `cxface.hed` file defines `TRUE` and `SUCCESS` to be equal to 1, and `FAIL` and `FALSE` to be equal to 0.

Your C program will look something like this. (Note that this must be the first function definition in your source file, because after the code is loaded, execution starts at the "top" of the code, which will be the first function defined in the file.)

```
bool mycprim(argc,link)
 int argc;
 void (*link)();

 {
  ... variable declarations etc.

  set_link(link);
  if (argc != 2)
   return(FAIL);

  ... rest of program

  return(SUCCESS);
 }
```

The `argc` parameter is the Prolog argument count. This particular program is expecting 2 Prolog arguments, and it immediately fails if this requirement is not met. The body of the program will probably make calls to the Prolog term access routines such as `get_arg`, `get_tag` etc., and to the term construction functions to bind variables.

When the code is written, compile it. The MPW command is

```
C user.c
```

Then link the resulting `user.c.o` with `xface.o` to produce the final code.
For example, in MPW:

```
link -rt MINE=100 user.c.o xface.o -o MyResource
```

The `-rt` option declares the resource type to be `MINE` and the resource ID to be `100` (you may choose any resource type and ID). The resulting code resource will be in the file `MyResource`.

The code may then be run from MacPROLOG™ using the `call_c` primitive. Enter the clause:

```
mycprim(A,B):-
        call_c(A,B,'MINE',100).
```

After calling `res_open` to open the file `MyResource`, you can make calls to `mycprim` just as if it was an ordinary Prolog program.

Note

It is important that you specify `user.c.o` as the *first* file in the link sequence. If you do not, then the code's entry point will be somewhere other than you intended!
Similarly, if you call other C functions, they must be defined in your source file *after* the main function which will be called from Prolog.

21.9.2 Using `call_c`

Suppose that you have a C code resource with resource type `MINE` and resource ID of `100`, in a file called `MyResource`. In MacPROLOG, choose a name for the C primitive, say `mycprim`. Add a clause of the form:

```
mycprim(A1, A2, ..., Ak):-
        call_c(A1, A2, ..., Ak, 'MINE', 100).
```

If your C code handles different numbers of arguments, add such a clause for each arity. For example, if your C routine has a 2 argument form and a 3 argument form, you would add the following clauses:

```
mycprim(A,B):- !,
        call_c(A,B,'MINE',100).
mycprim(A,B,C):-
        call_c(A,B,C,'MINE',100).
```

Before calling `mycprim`, open the resource file containing the code:

```
res_open('MyResource').
```

Then you can call `mycprim` just as if it was an ordinary Prolog program.

21.10 Technical Tips

21.10.1 Memory Usage and Global Variables

Since your code is written as a separate resource, you cannot use global (such as *static*) variables. You may only use local variables. If you do try to use global variables, you may not get any compiler errors or warnings, but when you run the code you will be unwittingly overwriting the MacPROLOG™ application globals. This is not a good idea!
If you need to use global variables, we suggest that you allocate a block of memory in your C code (using the Mac Memory Management routines), and pass the memory's handle as a parameter of the primitive. You could use the property handling primitives of MacPROLOG™ to store the handle.

For example, in Prolog, your definition of `'<LOAD>'` might include the following

```
...
res_open('MyResource'),        %open resource file
set_prop(myprim,memhandle,0),  %initialise memhandle property
...
```

When you first call the external primitive you pass the handle NIL. The C code can examine the handle, allocate memory if it is NIL, or otherwise simply use the memory indicated by the handle.

Suppose `mycprim` would normally take two arguments. Your C code could look for third and fourth arguments, the third being the memory handle, and the fourth a variable which the C code will instantiate to the (possibly changed) memory handle on returning to Prolog.

The program for calling the C primitive might be as follows.

```
mycprim(A,B) :-
        get_prop(myprim,memhandle,Handle),
        call_c(A,B,Handle,Newhandle,Result,'MINE',100),
        set_prop(myprim,memhandle,Newhandle),
        Result is 1.
```

This method allows for the handle to change during execution of the C primitive (perhaps the memory block needs to grow, or be freed). However, because we always need to record the new handle, we cannot let the call to `call_c` fail. This means that the C code must always return `SUCCESS`. However, the success or failure of `mycprim` can be effected by adding another argument, `Result`, which can be bound to either 0 or 1 by the C code to indicate the failure or success.
(The only other way that `call_c` can fail is if for some reason the code resource cannot be loaded. In this case `Handle` will not be used anyway.)

21.10.2 Writing more than one C Primitive

Each code resource has only one entry point. and therefore only one C function can be called from any given code resource. If you want to have more than one C primitive, one way to do it is to have several resource files, each containing a single C program. However, this is not ideal, and a neater way of writing several C primitives is to use one of the following methods.

<u>Method 1</u>

You can combine several code resources into one file.

Suppose you have files `user.c`, `user2.c` and `user3.c` which each contain a C primitive. After compiling each of the files, the following MPW commands will combine these into a single file.

```
link -rt CCCA=0 user.c.o xface.o -o MyResource
link -rt CCCB=0 user1.c.o xface.o -o MyResource
link -rt CCCC=0 user2.c.o xface.o -o MyResource
```

The file `MyResource` will contain three separate code resources of types `CCCA`, `CCCB` and `CCCC` respectively.

Suppose these have arities of 1, 2 and 3, then the Prolog calls would be as follows:

```
myprim1(A) :-
        call_c(A,'CCCA',0).
myprim2(A,B) :-
        call_c(A, B, 'CCCB',0).
myprim3(A,B,C) :-
        call_c(A, B, C, 'CCCC',0).
```

NOTE In MPW you cannot link more than one code resource of the same *resource type* into one file. The MPW Linker deletes any code resources matching the given type before creating the new one, even if they have different resource ID's. This is why in the above example we have used the types `CCCA`, `CCCB` and `CCCC` rather than using one type and three different ID's.

Method 2

You can create the effect of different primitives within a single code resource by passing a 'selector' flag from Prolog through to a single C function, and branching in the C code depending on the value of this flag. Such a C primitive might look something like this:

```
bool myprim(argc,link)
     short argc;
     void (*link)();
     {
      set_link(link);
      switch (get_int_val(get_arg(1)))
             {
             case 1:
                return(prim1(argc));
             case 2:
                return(prim2(argc));
             ... etc.

             }
     }

bool prim1(argc)
     short argc;
     {
       if (argc != 3)
             return(FAIL);
     ... etc.
     }

bool prim2(argc)
     short argc;
     {
       if (argc != 2)
             return(FAIL);
     ... etc.
     }
```

The corresponding calls from Prolog would be:

```
     myprim1(X,Y):-
             call_c(1,X,Y,'MINE',100).
     myprim2(X):-
             call_c(2,X,'MINE',100).
     ... etc.
```

21.10.3 Storing Code Resources in Prolog Files

A Prolog program (source or object code) is stored in the *data fork* of a file. This means that you can add resources (in particular, code resources) to it without affecting the Prolog code. It may be convenient to store your C code resources in the same file as the Prolog program that calls them. To do this, simply give the Prolog file name as the output file name when linking.

21.11 C Primitive Example

The example given below demonstrates how to use some of the interface functions in writing a C primitive. The primitive is not a very useful one (in fact the same effect can be achieved using existing Prolog primitives), but it serves to demonstrate how to manipulate Prolog terms in C.

Our primitive will take three arguments. The first may be either a constant, a list or a tuple (remember that a tuple is the MacPROLOG internal representation of a functor term). The second and third arguments must be variables.
The second argument will be bound to the length of the first argument.
If the first argument is a constant, the third argument will become the text of the constant reversed. Otherwise a list will be converted to a tuple, and vice versa.

The name of our new primitive in Prolog will be `ctest`.

The call

```
ctest(apples, X, Y)
```

will bind `X` to `6`, and `Y` to the atom `selppa`.

The call

```
ctest([apples,pears,bananas], X, Y)
```

will bind `X` to `3`, and `Y` to the term `apples(pears,bananas)`.

The call

```
ctest(foo(1,2,3), X, Y)
```

will bind `X` to `4`, and `Y` to the list `[foo, 1, 2, 3]`.

21.11.1 C Source Code

This is the source of the new C primitive.

```
/* user.c
 * Sample C Primitive for LPA MacPROLOG™
*/

#include "cxface.hed"                   /* Interface declarations */

bool test(argc,link)
 short argc;
 void (*link)();
 {
  cellpo t1,t2,t3;
  int i,len;
  char str1[255],str2[255];

  set_link(link);                       /* "glue" us to MacPROLOG */
  if (argc != 3)
    return(FAIL);                       /* Check arg. count */
  t1 = get_arg(1);                      /* Read Prolog arguments */
  t2 = get_arg(2);
  t3 = get_arg(3);
  if ((get_tag(t2) != VARTAG) || (get_tag(t3) != VARTAG))
    return(FAIL);                       /* Fail if not vbles. */

  switch (get_tag(t1))                  /* Look at type of 1st arg. */
  {

   case CONTAG:
   case STRTAG:
    len = get_con_len(t1);              /* Get length */
    t2 = put_int_val(t2,len);           /* Bind 2nd arg. to length */
    get_con_text(str1,t1);              /* Get text */
    for (i=0; i<len; i++)               /* Reverse string */
      str2[i] = str1[len-1-i];
    str2[len] = '\0';                   /* Terminate string */
    t3 = put_con_text(t3,str2);         /* Bind 3rd arg */
    return(SUCCESS);                    /* Return to Prolog */

   case LISTTAG:
    len = get_list_len(t1);             /* Get length */
    put_int_val(t2,len);                /* Bind 2nd arg. to length */
    put_tpl(t3,len);                    /* Make 3rd arg a tuple */
    for (i=1; i<=len; i++)              /* Put list els.into tuple */
     {
      put_copy_cell(get_tpl_nth(i,t3),get_list_head(t1));
      t1 = get_list_tail(t1);           /* Walk through list */
     }
    return(SUCCESS);
```

```
  case TPLTAG:
   len = get_tpl_len(t1);           /* Get length */
   put_int_val(t2,len);             /* Bind 2nd arg. to length */
   for (i=1; i<=len; i++)           /* Create list ... */
    {
     put_list(t3);
     put_copy_cell(get_list_head(t3),get_tpl_nth(i,t1));
     t3 = get_list_tail(t3);
    }
   put_nil(t3);                     /* Terminate list */
   return(SUCCESS);

  default:
   return(FAIL);                    /* Fail if not correct type */
 }
};
```

Note the methods used for creating and reading lists and tuples. In particular, to create the list above, we

i) construct a LISTTAG cell:

```
put_list(t3);
```

ii) bind its head term to a copy of the ith tuple cell:

```
put_copy_cell(get_list_head(t3),get_tpl_nth(i,t1));
```

iii) and get a pointer to the tail cell

```
t3 = get_list_tail(t3);
```

This will then be bound to another LISTTAG cell at the top of the next loop, and so on.

iv) To terminate the list we must make the final 'tail' a NILTAG cell:

```
put_nil(t3);                 /* Terminate list */
```

21.11.2 Compiling, Linking and Running

When this source text has been entered, it may be compiled and linked with the MPW commands:

```
C user.c
link -rt TEST=0 user.c.o xface.o Testfile
```

Now Testfile contains the code resource ready for calling from MacPROLOG.

In Prolog, enter the following definition of ctest:

```
ctest(X, Y, Z):-
        call_c(X, Y, Z, 'TEST', 0).
```

After opening the resource file:

```
res_open('Testfile')
```

ctest may be called and will exhibit the behaviour described above.

21.12 Technical Specification and Calling Conventions

The C interface files and functions have been written according to the following conventions.

21.12.1 Data Types

A `short int` occupies 16 bits.
An `int` occupies 32 bits.
A floating point number of type `extended` occupies 80 bits.
Strings are terminated with a null character (`'\0'`).

21.12.2 Interface Functions Calling Conventions

All the interface functions (such as `get_int_val`, `put_real_val` etc.) are declared to be '`pascal`' type functions, which means they conform to the following calling conventions (this corresponds to the conventions used by the Macintosh Toolbox routines).

Parameters are evaluated from left to right, and are pushed onto the stack in that order. Space for the function result is reserved on the stack by the caller before pushing any parameters. The callee is responsible for removing the parameters.

A `short int` is passed as a 16 bit value.
An `int` is passed as a 32 bit value.
An `extended` number is passed as a 32 bit pointer to an 80 bit value. When an `extended` number is to be returned as a function result (e.g. in `get_real_val`), the caller must allocate space for the 80 bit value and push a 32 bit pointer to this buffer in the place of the function result (i.e. this pointer is pushed on the stack before the other parameters to the function call).
A string pointer is passed as a 32 bit value.

21.12.3 C Calling Conventions

A C function's parameters are evaluated from right to left and are pushed on the stack in that order.

The caller is responsible for removing the parameters from the stack. Parameters are pushed as described above except that a `short int` is sign extended and pushed as a 32 bit value.

The function result is returned in registers rather than on the stack. In particular, the boolean result of a C primitive is expected in the low byte of register D0.

21.12.4 Examples

1. When the Prolog `call_c` primitive calls a C primitive, the following operations are performed. Recall that the C function has the form

```
bool cprim(argc,link)
short argc;
void (*link)();
```

i) Push the `link` parameter on to the stack as a 32 bit value
ii) Push the Prolog argument count as a 32 bit value, sign extended from a 16 bit value
iii) Call the C code
iv) Clear from the stack the two 32 bit parameters to the C call
iv) Read the C function result in the low byte of D0. If it is zero, fail the Prolog call, otherwise succeed the call.

2. Consider the `put_int_val` function, which is declared as

```
pascal cellpc put_int_val(cp,val)
cellpc cp;
int val;
```

The code for this function is written in assembly language and does the following operations.

i) Pop the return address from the stack
ii) Pop the `val` parameter as a 32 bit value
iii) Pop the `cp` parameter as a 32 bit value
iv) Push the return address
v) Create the Prolog term, bind it to the cell at `cp` etc.
vi) Move the 32 bit `cellpc` result to just above the return address (i.e. to Stack Pointer + 4)
vii) Return to C

22 The Pascal Interface

22.1 Introduction

MacPROLOG™ allows you to write your own primitives in Pascal and call them directly from within Prolog. The primitives must be compiled as separate code resources, and they are then loaded as they are needed by the Macintosh Resource Manager. This means that there is no need to rebuild the LPA MacPROLOG™ application file, but you can create your own separate library of Pascal routines which are callable from Prolog. The extent to which these new primitives behave like a program written in pure Prolog depends on how many of the primitive's uses have been catered for by the Pascal programmer.

A Pascal primitive may have arguments passed to it from Prolog. There is a collection of interface routines which allow you to access these arguments and their values, and return values to Prolog by binding the variables of the call.

Like a Prolog program, a Pascal primitive either succeeds or fails, possibly binding variables in the process. Unlike a Prolog program, however, the Pascal primitive cannot be non-deterministic: that is, you cannot backtrack into it to evaluate alternative solutions. If you want this behaviour, you can write a Prolog program to sequence through non-deterministic solutions to your Pascal primitive.

Having written your new primitive as a code resource, it is called from within Prolog with the `call_pascal` primitive, which will load and run the specified code. Prolog arguments to the call may be accessed from within the Pascal code.

22.1.1 Requirements

To use the Pascal Interface, you will need the supplied library of Prolog interface functions, a Pascal Compiler and Linker, and a 68000 Assembler.

Important Note
LPA MacPROLOG™ was written using the MPW (Macintosh Programmers' Workshop) package, and instructions for creating code resources here refer to MPW. If you use a different compiler, there may be differences in function calling conventions, and you may have to make some adjustments to the supplied source files or to the way in which you call the Prolog interface functions. See the Technical Specification Section for further details.

22.2 Calling a Pascal Function - An Outline

Here we sketch the format of a Pascal primitive and the method of calling it from Prolog. For a more detailed description, refer to later sections of this chapter. In particular, a complete example is given at the end of this chapter.

When a Pascal program is called from Prolog it will be called with two arguments.

The first argument (`argc` in the example below) is an integer giving the number of Prolog arguments to the call. (You can read these Prolog arguments using the `get_arg` function, described later in the Term Access Functions section.)

The second argument is a pointer to a routine in the MacPROLOG™ application code which the various Prolog term access routines use. This second argument must be passed on to the `set_link` procedure (defined in the interface files) as the first call of your Pascal code; thereafter this `link` argument may be ignored. The `set_link` procedure does the "gluing" of your code resource to MacPROLOG™.

The Pascal primitive must return a Boolean value corresponding to the primitive succeeding or failing. The interface file defines `SUCCESS` to be equal to `TRUE`, and `FAIL` to be equal to `FALSE`.

Your Pascal program will look something like this.

```
function myprim(argc:integer;link:longint):boolean;

  ... variable declarations etc.

  BEGIN
        set_link(link);
        if argc <> 3 then
          myprim := FAIL;

        ... rest of program

        myprim := SUCCESS;
  END;
```

As described above, `argc` is the Prolog argument count. This particular program is expecting 3 Prolog arguments, and it immediately fails if this requirement is not met. The body of the program will make calls to the Prolog term access routines to read the three Prolog arguments, and to the term construction functions which will bind any variables of the call.

The Pascal code must then be compiled and linked as a code resource (you may give it any resource type and ID). It is then called from MacPROLOG using the `call_pascal` primitive (see below).

22.3 **call_pascal - load and run a Pascal code primitive.**

call_pascal(A_1, A_2, . . . , A_k, *restype*, *resid*)

ARGUMENTS

A_1,A_2, ...,A_k	: Prolog arguments to the Pascal primitive
restype	: resource type of code resource
resid	: resource ID of code resource

This call loads and runs a Pascal routine in the code resource identified by the *restype* and *resid* arguments. The first *k* arguments to `call_pascal` will be placed in the Prolog argument registers for subsequent access by the Pascal primitive.

Note that the resource file containing the resource will *not* be automatically opened by `call_pascal`: you should do this with a call to `res_open` before you call the code.

The `call_pascal` primitive will succeed or fail depending on the `SUCCESS`/`FAIL` result passed back from the Pascal primitive. `call_pascal` will also fail if the resource cannot be loaded for any reason. This would happen if, for example, the resource file was not open.

EXAMPLE

Suppose you have a Pascal program in a code resource of type `MINE` and with ID `100`, which expects 3 arguments. If in Prolog you name this primitive `pascprim`, then you would add the following clause.

```
pascprim(A,B,C) :-
   call_pascal(A,B,C,'MINE',100).
```

After you have opened the resource file containing your code resource (by calling `res_open`), `pascprim` can be called just like any other Prolog program.

22.4 Storage of Terms in MacPROLOG

In order to make use of the functions providing an interface to Prolog, you need to be aware of how MacPROLOG stores terms internally. You do not need to know *exact* memory layouts but you need to know about the various term structures and how they are represented. The following discussion in this section describes the general storage mechanism for Prolog terms; the next section gives details about individual data types.

22.4.1 Tagged Cells

A term in MacPROLOG is stored in a *tagged cell*. The *tag* of the cell indicates the type of data stored in the cell, and the rest of the cell represents its value.

Access to these cells from Pascal is always via *cell pointers*. The type `cellpo` is defined in the Pascal interface to be a tagged cell pointer.

All term manipulation is done by passing cell *pointers* to the various term access and construction functions provided. You will never manipulate the tagged cells directly.

The Pascal interface provides a number of these term access functions which allow a Pascal program to find out about the term stored in a particular cell. For example, the `get_int_val` function returns the integer value of an integer cell. There are also a number of term construction functions which allow new Prolog terms to be built and returned as bindings for existing unbound variables. For example, the `put_list` function binds a variable to a Prolog list structure. These functions are described in more detail later.

These term access and construction functions present a fairly abstract view of the MacPROLOG data structures, and therefore you need not be over concerned with the exact internal representation and memory layout of Prolog terms.

22.4.2 Prolog Argument Registers

A Pascal function can access the Prolog arguments given in the call to `call_pascal`. There are 31 Prolog argument registers each of which can hold any Prolog term (i.e. a tagged cell). There may therefore be up to 31 Prolog arguments to the Pascal function which will be placed in these argument registers. The interface function `get_arg` can then be used from within Pascal to read these arguments. `get_arg` will return a pointer to the term cell in a specified Prolog argument register (see the Term Access Functions section below).

22.5 MacPROLOG Term Structures

There are seven different term types in LPA MacPROLOG™. These are detailed below. As described above, each Prolog term is held in a tagged cell. The tag values (defined in the interface files) are as follows.

TERM TYPE	TAG
Variable	VARTAG
Integer	INTTAG
Real	REALTAG
Constant	CONTAG or STRTAG
List	LISTTAG
Empty List	NILTAG
Tuple	TPLTAG

22.5.1 Variable

A `VARTAG` cell represents an unbound variable. Such cells may be bound to other Prolog terms using the term construction primitives such as `put_int_val`, `put_con_text` etc.

22.5.2 Integer

An `INTTAG` cell represents an integer, stored as a 24 bit signed value. Note that Pascal uses 32 bits for integers, but integers outside the range -8388608 to 8388607 will not be correctly represented in Prolog.

22.5.3 Real

A `REALTAG` cell represents a floating point number. It is stored as an 80 bit value, which corresponds to the Pascal data type `extended`.
(Note that in MacPROLOG™ real numbers are automatically converted to integers if possible, so that when you construct a real cell you may sometimes get an integer cell instead).

22.5.4 Constant

A `CONTAG` or `STRTAG` cell represents some text stored as a sequence of ASCII characters. The cell may have either of the two tags `CONTAG` or `STRTAG`, which merely reflects whether the text is stored as a 'permanent' dictionary entry or as a 'temporary' object in the Prolog 'heap'. The actual format of the stored text is identical in each case.

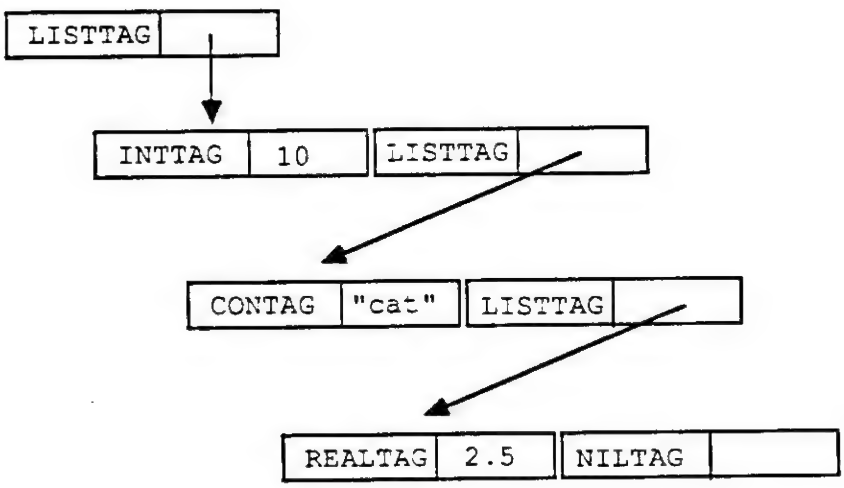
22.5.5 List

A LISTTAG cell represents a Prolog list structure. The list cell actually points to two more term cells, one representing the term at the head of the list, and the other representing the tail. The tail will usually be another LISTTAG cell, or the empty list (i.e. a NILTAG cell).

For example, the Prolog list

```
[10, cat, 2.5]
```

is represented by a LISTTAG cell which points to an integer cell with the value 10, and another LISTTAG cell. This second list cell points to a constant cell with the value cat, and a third list cell, which in turn points to a real cell with value 2.5, and a NILTAG cell.



Thus the internal representation of the above list is

```
[10, [cat, [2.5, []] ] ]
```

22.5.6 Empty List

A NILTAG cell represents the empty list. As described above, one of its uses is as a 'tail cell' to terminate a list. A NILTAG cell has no "value".

22.5.7 Tuple

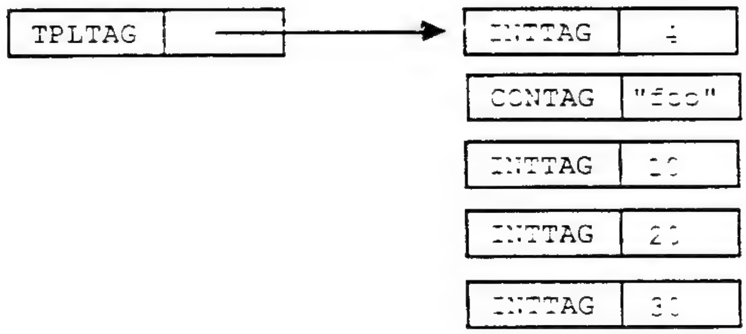
A TPLTAG cell represents a Prolog tuple structure. A tuple is a fixed size array of elements, where each element is any Prolog term. This corresponds to the internal representation of a functor and its arguments.

For example, the Prolog term

```
foo(10,20,30)
```

is represented internally as a tuple with four elements, the first element being a constant cell with value foo, and the remaining elements being integer cells with values 10, 20 and 30. (MacPROLOG also stores an extra "size" cell at the beginning of the tuple structure which gives the number of tuple elements.)

The above tuple therefore looks like this.



Since a tuple is a fixed size, it has a more compact representation than a list.

22.6 Pascal Interface Term Access Functions

These functions allow you to read Prolog terms from within Pascal. As mentioned above, a Prolog term is stored in a tagged cell, but all term manipulation is done using *pointers* to these cells.

All these functions are declared in the file `pxface.hed` (which you will include in your Pascal source file).

Note that these functions will do *no* checking of arguments: you are responsible for ensuring that any arguments you pass to one of these routines are of the correct type. For example, if you pass a `LISTTAG` cell pointer to the function `get_con_text`, there is no guarantee of what the results might be!

The types `cellpo` (a tagged cell pointer), `txt` (a 255 character string) and `txtpo` (a string pointer) are declared in the file `pxface.hed`.

22.6.1 `get_arg`

```
function get_arg(num:integer):cellpo;
```

This function returns a pointer to the cell in Prolog argument register `num`, where `num` ranges from 1 to `argc` (the Prolog argument count, passed as a parameter to your Pascal function). You may read the Prolog argument registers in any order. However, there is no guarantee that you will get anything sensible if you try to read a register beyond `argc`. (There are no checks done by `get_arg` on the validity of `num`.)

22.6.2 `get_tag`

```
function get_tag(cp:cellpo):integer;
cellpo cp;
```

This function returns the tag of a Prolog cell pointed to by `cp`. The value returned will be one of:

```
VARTAG INTTAG REALTAG CONTAG STRTAG LISTTAG NILTAG TPLTAG
```

The tag indicates the type of the data stored in the cell.

22.6.3 `get_int_val`

```
function get_int_val(cp:cellpo):longint;
```

This function returns the integer value of an `INTTAG` cell pointed to by `cp`. As described above, although a Pascal `longint` type is returned, only the lower 24 bits are used by MacPROLOG. (The integer returned by this function will have been sign extended to 32 bits).

22.6.4 `get_real_val`

```
function get_real_val(cp:cellpo):extended;
```

This function returns the real (floating point) value of a `REALTAG` cell pointed to by `cp`.

22.6.5 `get_con_text`

```
function get_con_text(str:txtpo;cp:cellpo):txtpo;
```

This function returns the text of either a `CONTAG` or `STRTAG` cell pointed to by `cp`. The text will be copied as a Pascal string into the buffer pointed to by `str`; this pointer is also returned as the function result.
You must ensure that `str` points to a string buffer large enough to hold the text: the `get_con_text` function does not allocate any space for you.

22.6.6 `get_con_len`

```
function get_con_len(cp:cellpo):longint;
```

This function returns the actual number of characters in the text of the `CONTAG` or `STRTAG` cell pointed to by `cp`.

22.6.7 `get_list_head`

```
function get_list_head(cp:cellpo):cellpo;
```

If `cp` points to a `LISTTAG` cell, this function returns a pointer to the cell representing the term at the head of the list.

22.6.8 `get_list_tail`

```
function get_list_tail(cp:cellpo):cellpo;
```

If `cp` points to a `LISTTAG` cell, this function returns a pointer to the cell representing the tail of the list.

22.6.9 get_list_len

```
function get_list_len(cp:cellpo):longint;
```

If cp points to a LISTTAG cell, this function returns the length of the list (i.e. the number of elements in the list).

22.6.10 get_tpl_len

```
function get_tpl_len(cp:cellpo):longint;
```

If cp points to a TPLTAG cell, this function returns the number of elements in the tuple.

22.6.11 get_tpl_nth

```
function get_tpl_nth(nth:longint;cp:cellpo):cellpo;
```

If cp points to a TPLTAG cell, this function returns a pointer to the cell representing the nth element of the tuple. The first element of the tuple is numbered 1. The value of nth should not exceed the length of the tuple.

22.7 Pascal Interface Term Construction Functions

The following functions allow you to construct new Prolog terms and to bind existing unbound variables to them. Each function expects to be passed a pointer to an unbound variable (i.e. a pointer to a VARTAG cell), which will then be instantiated to a new Prolog term. The function returns a pointer to this newly created term.

In a simialy way to the term access routines, no check is made that the cell pointer does in fact point to a VARTAG cell.

These bindings will automatically be undone on backtracking.

You should not try to construct a new term in a cell which is not a variable, as this is not logical and may lead to strange results.

22.7.1 put_int_val

```
function put_int_val(unb:cellpo;val:longint):cellpo;
```

This function constructs an integer cell and binds it to the variable pointed to by unb. As mentioned previously, only the lower 24 bits of val are stored.

22.7.2 put_real_val

```
function put_real_val(unb:cellpo;val:extended):cellpo;
```

This function constructs a real cell and binds it to the variable pointed to by unb. As noted previously, if val can be expressed as an integer, an integer cell will be constructed instead. Thus the tag of the constructed cell may be either REALTAG or INTTAG.

22.7.3 put_con_text

```
function put_con_text(unb:cellpo;str:txtpo):cellpo;
```

This function constructs a constant cell from the string pointed to by str and binds it to the variable cell pointed to by unb. The tag of the returned cell will be either CONTAG or STRTAG.

22.7.4 put_list

```
function put_list(unb:cellpo):cellpo;
```

This function constructs a list cell and binds it to the variable pointed to by unb. Two extra cells are created, one for the head of the list and one for the tail. Both of these will be initialised to unbound variables. You can access these cells (and bind new Prolog terms to them) using the get_list_head and get_list_tail functions.

22.7.5 `put_nil`

```
function put_nil(unb:cellpo):cellpo;
```

This function constructs an empty list cell (a `NILTAG` cell) and binds it to the variable pointed to by `unb`.

22.7.6 `put_tpl`

```
function put_tpl(unb:cellpo;size:longint):cellpo;
```

This function constructs a tuple cell and binds it to the variable pointed to by `unb`. An array of extra cells is created for the `size` tuple elements. All of these will be initialised to unbound variables. You can access these cells (and bind new Prolog terms to them) using the `get_tpl_nth` function.

22.7.7 `put_copy_cell`

```
function put_copy_cell(dest,src:cellpo):cellpo;
```

This function copies the term pointed to by `src` to the term pointed to by `dest`. The function returns a pointer to the `dest` term.
The destination cell must be an unbound variable at the time of the call, but the source cell may be *any* Prolog term (including another unbound variable). If you use this function to copy one variable cell to another, the effect is to make both cells refer to the same variable, and binding one of these variables to a new Prolog term automatically binds the other variable to the same term.

22.8 Reporting Errors from a Pascal Primitive

If you wish to report an error from within your Pascal code, you may call the procedure `errortrap` with an error code signalling the type of the error.

22.8.1 `errortrap`

```
procedure errortrap(errno:integer);
```

This procedure generates a Prolog numbered error. It does *not* return control to the calling routine, but jumps straight to the Prolog error handler. The error will be handled in exactly the same way as any other Prolog error. See the chapter on Implementing an Application for more details on error handling, and the Appendix for details of the error numbers available.

22.9 Creating a Pascal Primitive

NOTE This documentation assumes you are using the Macintosh MPW Pascal Compiler, 68000 Assembler, and Linker. If you are using a different Compiler and Linker you may need to refer to your own manuals for specific instructions on creating code resources.
See the Technical Specification section to determine what changes, if any, you may need to make to the Pascal or Assembly language source code provided with MacPROLOG.

To create a new Pascal primitive for MacPROLOG you will need the files `xface.o`, `pxface.hed` and `user.p`.

`xface.o` is an object code file containing the routines giving access to the Prolog data structures. It is written in assembly language, and needs to be linked in with your Pascal object code to create the final code resource.
NOTE The source for `xface.o` is in the file `xface.a`, which you will only need to use if you find that the format of `xface.o` is not compatible with your Linker. In this case, you will need to assemble `xface.a` using an appropriate 68000 Assembler, (first making any textual changes to the source to conform to your assembler's syntax and other features) before linking the resulting object code with your Pascal object code file.

`pxface.hed` is a text file to be included in your Pascal source code file, `user.p`. It contains all the type and function declarations needed to interface with Prolog.

`user.p` is a text file which will contain the source for your Pascal primitive.

22.9.1 The Format of a Pascal Primitive

When a Pascal program is called from Prolog it will be called with two arguments.

The first argument is an integer giving the number of Prolog arguments to the call. (You can read these Prolog arguments using the `get_arg` function, described in the Term Access Functions section.)
The second argument is a pointer to a routine in the MacPROLOG™ application code which the various Prolog term access routines use. All you need to do with this second argument is pass it on to the `set_link` procedure as the first procedure call of your Pascal code; thereafter you should ignore this "link" argument. The `set_link` function does the "gluing" of your code to MacPROLOG™; if you omit to call `set_link` you can expect to have serious errors!

The Pascal primitive must return a `boolean` value corresponding to the primitive succeeding or failing. The `pxface.hed` file defines `SUCCESS` to be equal to `TRUE`, and `FAIL` to be equal to `FALSE`.

Your Pascal program will look like this. (Note that this must be the *first* function definition in your source file, because after the code is loaded, execution starts at the "top" of the code, which will be the first function defined in the file.)

```
function myprim(argc:integer;link:longint):bool;

... variable declarations etc.

BEGIN
set_link(link);
if argc <> 2 then
 myprim := FAIL;

... rest of program

myprim := SUCCESS;
END;
```

Note that `argc` is the Prolog argument count. This particular program is expecting 2 Prolog arguments, and it immediately fails if this requirement is not met. The body of the program will probably make calls to the Prolog term access routines such as `get_arg`, `get_tag` etc., and to the term construction functions to bind variables.

When you have written the code, compile it. In MPW, the command is

```
Pascal user.p
```

Then link the resulting `user.p.o` with `xface.o` to produce the final code.
For example, in MPW:

```
link -rt MINE=100 user.p.o xface.o -o MyResource
```

The `-rt` option declares the resource type to be `MINE` and the resource ID to be `100` (you may choose any resource type and ID). The resulting code resource will be in the file `MyResource`.

The code may then be run from MacPROLOG™, after calling `res_open` to open the file `MyResource`, using the `call_pascal` primitive:

```
myprim(A,B,C):-
        call_pascal(A,B,C,'MINE',100).
```

NOTE

It is important that you specify `user.p.o` as the *first* file in the link sequence. If you do not, then the code's entry point will be somewhere other than you intended!
Likewise, the Pascal primitive has to be the *first* function definition in the source file `user.p`.

21.8.1 Using call_pascal

Suppose that you have a Pascal code resource with resource type `MINE` and resource ID of `100`, in a file called `MyResource`. In MacPROLOG, choose a name for the Pascal primitive, say `myprim`. Add a clause of the form:

```
myprim(A1, A2, ..., Ak):-
        call_pascal(A1, A2, ..., Ak, 'MINE', 100).
```

If your Pascal code handles different numbers of arguments, add such a clause for each arity. For example, if your Pascal routine has a 2 argument form and a 3 argument form, you would add the following clauses:

```
myprim(A,B):-
        call_pascal(A,B,'MINE',100).
myprim(A,B,C):-
        call_pascal(A,B,C,'MINE',100).
```

Before calling `myprim`, open the resource file containing the code:

```
res_open('MyResource').
```

Then you can call `myprim` just as if it was an ordinary Prolog program.

22.10 Technical Tips

22.9.1 Memory Usage and Global Variables

Since your code is written as a separate resource, you cannot use global variables. You may only use local variables. If you do try to use global variables, you may not get any compiler errors or warnings, but when you run the code you will be unwittingly overwriting the MacPROLOG™ application globals. This is not a good idea!
If you need to use global variables, we suggest that you allocate a block of memory in your Pascal code (using the Mac Memory Management routines), and pass the memory's handle as a parameter of the primitive. You could use the property handling primitives of MacPROLOG™ to store the handle.

For example, in Prolog, your definition of `'<LOAD>'` might include the following

```
...
res_open('MyResource'),
set_prop(myprim,memhandle,0),
...
```

When you first call the Pascal primitive you pass the handle NIL. The Pascal code can examine the handle, allocate memory if it is NIL, or otherwise simply use the memory indicated by the handle.

Suppose myprim would normally take two arguments. Your Pascal code could look for third and fourth arguments, the third of which is the memory handle, and the fourth a variable which the Pascal code will instantiate to the (possibly changed) memory handle on returning to Prolog.

The program for calling the Pascal primitive might be as follows.

```
myprim(A,B):-
      get_prop(myprim,memhandle,Handle),
      call_pascal(A,B,Handle,Newhandle, Result,'MINE',100),
      set_prop(myprim,memhandle,Newhandle),
      Result is 1.
```

This method allows for the handle to change during execution of the Pascal primitive (perhaps the memory block needs to grow, or be freed). However, because we always need to record the new handle, we cannot let the call to call_pascal fail. This means that the Pascal code must always return SUCCESS. However, the success or failure of myprim can be achieved by adding another argument, Result, which can be bound to either 0 or 1 by the Pascal code.

(The only other way that call_pascal can fail is if for some reason the code resource cannot be loaded. In this case Handle will not be used anyway.)

22.9.2 Writing more than one Pascal Primitive

Each code resource has only one entry point and therefore only one function can be called for any given code resource. If you want to write several Pascal primitives you could have several files each containing a single code resource. However, a neater way of handling several Pascal primitives is to use one of the methods below.

Method 1

You can combine several code resources into one file.

Suppose you have files user.p, user2.p and user3.p each of which contains a Pascal primitive. After compiling each of the files, in MPW you could issue the following commands to combine these into a single file.

```
link -rt PASA=0 user.p.o xface.o -o MyResource
link -rt PASB=0 user1.p.o xface.o -o MyResource
link -rt PASC=0 user2.p.o xface.o -o MyResource
```

The file MyResource will contain three separate code resources of types PASA, PASB and PASC respectively.

Suppose these have arities of 1, 2 and 3, then the Prolog calls would be as follows:

```
myprim1(A):-
      call_pascal(A,'PASA',0).
myprim2(A,B):-
      call_pascal(A, B, 'PASB',0).
myprim3(A,B,C):-
      call_pascal(A, B, C, 'PASC',0).
```

Note In MPW you cannot link more than one code resource of the same type into one file. The MPW Linker deletes any code resources matching the given type before creating the new one, even if they have different resource ID's. This is why in the above example we have used the types PASA, PASB and PASC rather than using one type and three different ID's.

Method 2

You can create the effect of different primitives within a single code resource by passing a 'selector' flag from Prolog through to a single Pascal function, and branching in the Pascal code depending on the value of this flag. Such a Pascal primitive might look something like this:

```
function myprim(argc:integer;link:longint):boolean;
       BEGIN
        set_link(link);
        case (get_int_val(get_arg(1))) of
              1:
                 myprim := prim1(argc);
              2:
                 myprim := prim2(argc);
              ... etc.

        end;
       END;

function prim1(argc:integer):boolean;
       BEGIN
        if argc <> 3 then
              prim1 := FAIL;
        ... etc.
       END;

function prim2(argc:integer):boolean;
       BEGIN
        if argc <> 2 then
              prim2 := FAIL;
        ... etc.
       END;

... etc.
```

If the resource is of type `MINE` with ID `100`, the corresponding calls from Prolog would be:

```
       myprim1(X,Y):-
              call_pascal(1,X,Y,'MINE',100).
       myprim2(X):-
              call_pascal(2,X,'MINE',100).
       ... etc.
```

22.9.3 Storing Code Resources in Prolog Files

The source or object code of a Prolog program is stored in the *data fork* of a file. You can store your Pascal code resources in the resource fork of the same file if you wish. You may find it convenient to store Pascal code resources in the same file as the Prolog program that calls them. To do this, simply give the Prolog file name as the output file name when you link the code resources.

21.11 Pascal Primitive Example

The example given below demonstrates how to use some of the interface functions in writing a Pascal primitive. The primitive is not a very useful one (in fact the same effect can be achieved using existing Prolog primitives), but it serves to demonstrate how to manipulate Prolog terms in Pascal.

Our primitive will take three arguments. The first may be either a constant, a list or a tuple (remember that a tuple is the MacPROLOG internal representation of a functor term). The second and third arguments must be variables.
The second argument will be bound to the length of the first argument.
If the first argument is a constant, the third argument will become the text of the constant reversed.
Otherwise a list will be converted to a tuple, and vice versa.

The name of our new primitive in Prolog will be `pasctest`.

The call

```
pasctest(apples, X, Y)
```

will bind `X` to `6`, and `Y` to the atom `selppa`.

The call

```
pasctest([apples,pears,bananas], X, Y)
```

will bind `X` to `3`, and `Y` to the term `apples(pears,bananas)`.

The call

```
pasctest(foo(1,2,3), X, Y)
```

will bind `X` to `4`, and `Y` to the list `[foo, 1, 2, 3]`.

22.10.1 Pascal Source Code

This is the source of the new Pascal primitive.

```
(* user.p
 * Sample Pascal primitive for MacPROLOG™
 *)

unit MyPascal;

interface

function test(argc:integer;link:longint):boolean;

implementation

{$I pxface.hed}

function test(argc:integer;link:longint):boolean;

VAR
      t1,t2,t3,temp : cellpo;
      len, i : longint;
      text,rtext : txt;
      tpo : txtpo;

BEGIN
 set_link(link);
 if argc <> 3 then                              { Fail if not 3 args }
      test := FAIL;

 t1 := get_arg(1);                              { Read the args }
 t2 := get_arg(2);
 t3 := get_arg(3);

 if (get_tag(t2) <> VARTAG) | (get_tag(t3) <> VARTAG) then
      test := FAIL;                             { Fail if not vbles }

 case (get_tag(t1)) of                          { Look at 1st arg. }

  CONTAG, STRTAG:
    begin
      len := get_con_len(t1);                   { get length }
      t2 := put_int_val(t2,len);                { bind 2nd arg }
      tpo := get_con_text(@text,t1);            { get text }
      for i := 1 to len do                      { reverse string }
        rtext[i] := text[len+1-i];
      rtext[0] := char(len);                    { put length in }
      t3 := put_con_text(t3,@rtext);            { bind 3rd arg. }
      test := SUCCESS;                          { return to Prolog }
    end;
```

```
LISTTAG:
  begin
   len := get_list_len(t1);              { Get length }
   t2 := put_int_val(t2,len);            { Bind 2nd arg }
   t3 := put_tpl(t3, len);               { Make 3rd arg a tuple }
   for i:=1 to len do                    { Copy list elts. in }
     begin
     temp:=put_copy_cell(get_tpl_nth(i,t3),get_list_head(t1));
     t1 := get_list_tail(t1);            { Walk thro' list }
     end;
   test := SUCCESS;
  end;


TPLTAG:
  begin
   len := get_tpl_len(t1);               { Get length }
   t2 := put_int_val(t2,len);            { Bind 2nd arg }
   for i:=1 to len do                    { Create list ... }
     begin
     t3 := put_list(t3);
     temp:=put_copy_cell(get_list_head(t3),get_tpl_nth(i,t1));
     t3 := get_list_tail(t3);            { Walk thro' list }
     end;
   t3 := put_nil(t3);                    { Terminate list }
   test := SUCCESS;
  end;


otherwise         { Fail if arg not correct type }
  test := FAIL;

end;      { of CASE statement }

END;  { of test function }

END.  { of source }
```

Note the methods used for creating and reading lists and tuples. In particular, to create the list above, we

i) construct a `LISTTAG` cell:
```
t3 := put_list(t3);
```

ii) bind its head term to a copy of the `i`th tuple cell:
```
temp:=put_copy_cell(get_list_head(t3),get_tpl_nth(i,t1));
```

iii) and get a pointer to the tail cell
```
t3 := get_list_tail(t3);
```

This will then be bound to another `LISTTAG` cell at the top of the next loop, and so on.

iv) To terminate the list we must make the final 'tail' a `NILTAG` cell:
```
t3 := put_nil(t3);                    { Terminate list }
```

22.10.2 Compiling, Linking and Running

When this source text has been entered, it may be compiled and linked with the MPW commands:

```
pascal user.p
link -rt TEST=0 user.p.o xface.o Testfile
```

Now `Testfile` contains the code resource ready for calling from MacPROLOG.

In Prolog, enter the following definition of `pasctest`:

```
pasctest(X, Y, Z):-
        call_pascal(X, Y, Z, 'TEST', 0).
```

After opening the resource file:

```
res_open('Testfile')
```

`pasctest` may be called and will exhibit the behaviour described above.

22.11 Technical Specification and Calling Conventions

The Pascal interface files and functions have been written according to the following conventions.

22.11.1 Data Types

An `integer` occupies 16 bits.

A `longint` occupies 32 bits.

A floating point number of type `extended` occupies 80 bits.

Strings are stored as a sequence of ASCII codes each occupying one byte, with a length byte at the beginning.

22.11.2 Interface Functions Calling Conventions

All the interface functions (such as `get_int_val`, `put_real_val` etc.) conform to the following calling conventions. (This corresponds to the conventions used by the Macintosh Toolbox routines).

Parameters are evaluated from left to right, and are pushed onto the stack in that order. Space for the function result is reserved on the stack by the caller before pushing any parameters. The callee is responsible for removing the parameters.

An `integer` is passed as a 16 bit value.

A `longint` is passed as a 32 bit value.

An `extended` number is passed as a 32 bit pointer to an 80 bit value. When an `extended` number is to be returned as a function result (e.g. in `get_real_val`), the caller must allocate space for the 80 bit value and push a 32 bit pointer to this buffer in the place of the function result (i.e. this pointer is pushed on the stack before the other parameters to the function call).

A string pointer is passed as a 32 bit value.

A `boolean` result is returned in the high byte of a 16 bit word.

22.11.3 Examples

1. When the Prolog `call_pascal` primitive calls a Pascal primitive, the following operations are performed. Remember that the Pascal primitive will be defined as

```
function prim(argc:integer;link:longint):boolean;
```

i) Push a 16 bit value on the stack to reserve space for the function result
ii) Push the Prolog argument count `argc` on to the stack as a 16 bit value
iii) Push the `link` parameter as a 32 bit value
iv) Call the Pascal code
v) Pop the Pascal function result as a 16 bit value. If the high byte is zero, fail the Prolog call, otherwise succeed the call.

2. Consider the `put_int_val` function, which is declared as

```
function put_int_val(cp:cellpc;val:longint):cellpc;
```

The code for this function is written in assembly language and does the following operations.

i) Pop the return address from the stack
ii) Pop the `val` parameter as a 32 bit value
iii) Pop the `cp` parameter as a 32 bit value
iv) Push the return address
v) Create the Prolog term, bind it to the cell at `cp` etc.
vi) Move the 32 bit `cellpo` result to just above the return address (i.e. to Stack Pointer + 4)
vii) Return to Pascal

23 Graphics in MacPROLOG

23.1 Introduction

Welcome to the MacPROLOG Graphics package. This package combines the Macintosh's sophisticated graphics with the programming power of MacPROLOG in a high level and declarative fashion. Graphic pictures, which can originate from inside MacPROLOG or be imported from external applications (MacDraw, MacPaint etc.), can be displayed both in **dialogues** and in a special type of window, the **graphic window**.

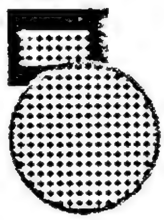
Pictures created within MacPROLOG are defined as terms of a **Graphic Description Language** (the GDL). This declarative language uses a natural, economic but powerful notation in which structured pictures are described as combinations and modifications of elementary pictures.

For example

```
crosses([thick(fillbox(0,0,30,40)),
                 greypen(pensize(2,2,filloval(20,0,60,60)))])
```

describes a *structured* picture comprising an overlapping box and oval, both filled with a crossed pattern, with the box having a thick black outline, and the oval a thinner grey outline.
The `fillbox` and `filloval` are *elementary* picture descriptors; and `pensize, thick,` and `greypen` are picture *transformers*.

The term above describes the picture :



Pictures can be imported from external applications through the clipboard or from resource files; in both these cases they are treated as elementary pictures with no internal structure.

Pictures are displayed in graphics windows using the primitive `add_pic`, which associates a named picture's description with a graphics window and simultaneously displays the picture in the window. The picture description is permanently associated with the window so that whenever the window is scrolled or needs refreshing, the picture can be redrawn. (You can also draw 'temporary' pictures by calling `draw_pic`).

Pictures can be displayed in dialogues by giving their picture description as the field descriptor for a `pbutton` or `pcheck` item of the dialogue (see the chapter on Advanced Dialogues).

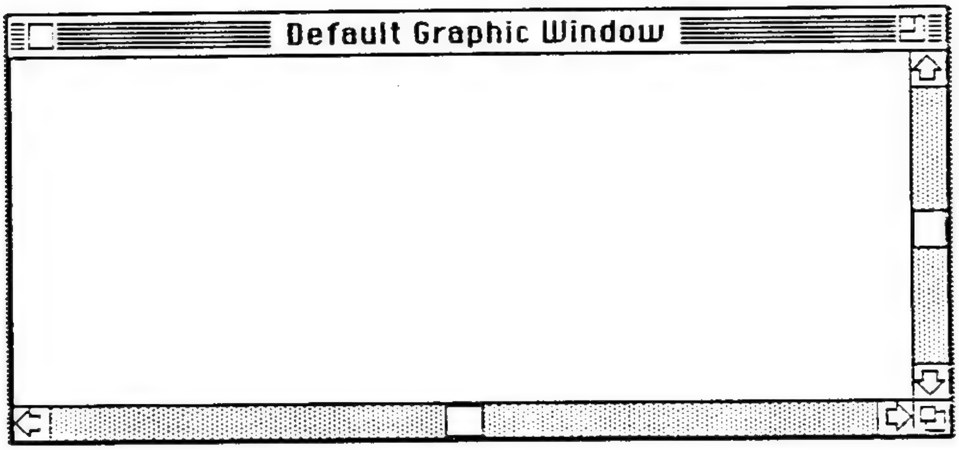
NOTE
The first time you use a graphics primitive in any MacPROLOG session, the **Graphics Boot** program must be loaded. This is done automatically for you, and a banner is displayed whilst it is being loaded. Thereafter all the graphics primitives remain resident in memory.

23.2 The Default Graphic Window

Using `add_pic` without naming a graphic window will associate the picture with the **Default Graphic Window**. This is a graphic window with Y coordinates ranging from -500 to +500 and X coordinates ranging from -500 to +500. Any pictures associated with this window can be scrolled either vertically or horizontally, as with all graphic windows.

Unlike most other graphic windows, it has no visible graphic tools, but it does have a selector associated with it, which allows selection of pictures in the window either by clicking over them with the mouse, or by dragging a grey rectangle around them. This latter method is a very common activity in graphics window, and we shall from here on refer to this as dragging a marqui.

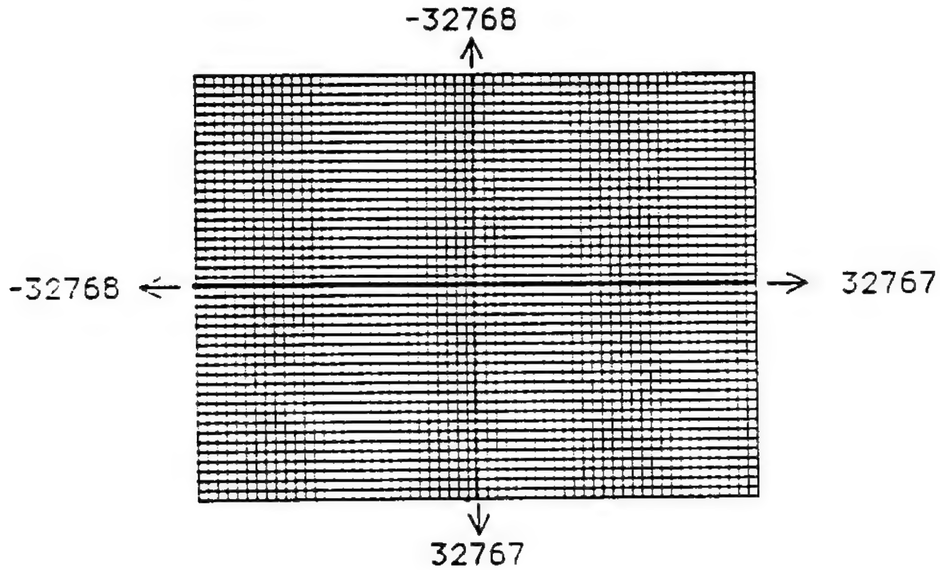
You can also edit the pictures in the window using the **Edit** menu commands (e.g. **Clear, Cut, Paste, Select all**).



23.3 Overview of QuickDraw

Graphics on the Macintosh is implemented by a set of low level graphics procedures called **QuickDraw**, an understanding of which will help in using MacPROLOG's graphics. This section gives a brief overview of the main ideas behind QuickDraw.

All information about location, placement, or translation is specified by the programmer in terms of coordinates on a plane. The coordinate plane is a two-dimensional grid, as shown below.



Coordinates must be integers in the range -32768 to +32767. The coordinate origin (0,0) is in the middle of the grid. As in traditional coordinate systems, the horizontal coordinates increase as you move from left to right, but, unusually for mathematical systems, the vertical coordinates increase as you move from top to bottom.

On the coordinate plane there are 4,294,967,296 unique points. Each point is at the intersection of a horizontal grid line and a vertical grid line. As the grid lines are infinitely thin, a point is infinitesimal.

Of course there are more points on this grid than there are dots on the Macintosh screen: when using QuickDraw you associate small parts of the grid with areas on the screen.

Shapes, lines, points and text are drawn in this large coordinate plane and are then mapped and clipped by the graphics system to the actual graphics window on the screen.

23.4 QuickDraw in MacPROLOG

What follows is a very brief discussion of how graphics is implemented in MacPROLOG: many details are necessarily omitted here, but full explanations and numerous examples are given in the chapters that follow.

23.4.1 Graphic Windows

In MacPROLOG, graphical objects are displayed in Graphic Windows. Each graphic window has associated with it a small part of the QuickDraw coordinate plane described above. You can specify how much of the coordinate plane you wish to use for each window - this is the window's "drawing area". Each graphic window displays on the screen a portion of its drawing area. You can roam around this drawing area using the window's scroll bars.

For example, as mentioned above, the **Default Graphic Window**'s drawing area ranges from -500 to +500 in each direction. This means that if you draw something at (1000,1000) in the window you won't ever see it! However, an object drawn at (300,300), say, can be brought into view on the screen using the window's scroll bars.

When you create a graphic window, (using the `wgcreate` primitive), you specify how large a drawing area you want for that window. You can subsequently change this size either programmatically or using the **Window details** dialogue from the **Windows** menu. In the Graphic Windows chapter there is a discussion of the effects of choosing different sizes of drawing area.

23.4.2 Coordinates, Points and Pixels

The drawing area of a graphic window is always centered on the coordinate plane at (0,0). When a new graphic window is created, the point (0,0) is displayed at the top left hand corner of the viewing pane.

Although a 'point' in QuickDraw is defined theoretically, you can think of points as equivalent to pixels.

In MacPROLOG, a point is specified as

```
(top, left)
```

where *`top`* is the vertical coordinate and *`left`* is the horizontal coordinate. (Remember that vertical coordinates increase in a *downward* direction.)

A rectangle is defined by specifying the position of its top left hand corner, together with its depth and width. Rectangles are represented in MacPROLOG by terms of the form

```
box(top, left, depth, width)
```

where *`(top, left)`* is the coordinate of the top left hand corner, and *`depth`* and *`width`* are the number of points in the vertical and horizontal directions respectively.

24 Graphic Description Language

A graphic window or dialogue can have any number of individual pictures. These are described by compound terms constructed from the various picture descriptors and transformers in MacPROLOG's own declarative **Graphic Description Language (GDL)**.

The graphic description language can be broken down into three main sections:

1) The **elementary picture descriptors** (e.g. `box, circle` etc).

These descriptors describe elementary pictures. They usually have arguments which are the coordinates of where the elementary picture is to be drawn. For example the term

```
box(10,20,30,40)
```

represents a box whose top left hand corner is at (10, 20) and, whose depth is 30 units and width 40 units. There are 17 predefined elementary picture descriptors and programmers can define new ones.

Note: The predefined picture descriptors are *not* the same as predefined predicates. Pictures cannot be constructed by direct calls to the picture descriptors. They are constructed by passing a GDL picture description as an argument to an `add_pic` call or to a dialogue construction call.

2) The **transformation descriptors** (e.g `trans, scale, thin`).

Transformation descriptors modify the description of a picture rather than describe a picture explicitly. They usually take as an argument a term describing the picture which is to be modified or transformed in some way.

For example, the transformational descriptor `grey` takes a picture as an argument and fills it with grey, so the term

```
grey(filloval(100,100,20,40))
```

represents an oval filled with a grey pattern (instead of the default black pattern). Other transformation descriptors allow pictures to be shifted and scaled, the pen's colour and size to be changed and the fill patterns to be modified. Transformation descriptors can be nested to an arbitrary depth.

3) Picture **aggregation.**

A picture can be described as an aggregate of several sub-pictures by grouping the sub-pictures into a list. When an aggregation is drawn, if the sub-pictures overlap, then the later sub-pictures are drawn on top of the earlier ones. For example the term

```
[box(10,20,30,40),grey(filloval(100,100,20,40))]
```

represents a box and a filled oval.

(Note: QuickDraw does not always correctly calculate the area enclosed by aggregations when they contain overlapping sub-pictures. This may lead to unexpected behaviour in selection and dragging tools. In particular it is not recommended to use aggregate pictures which contain totally overlapping sub-pictures.)

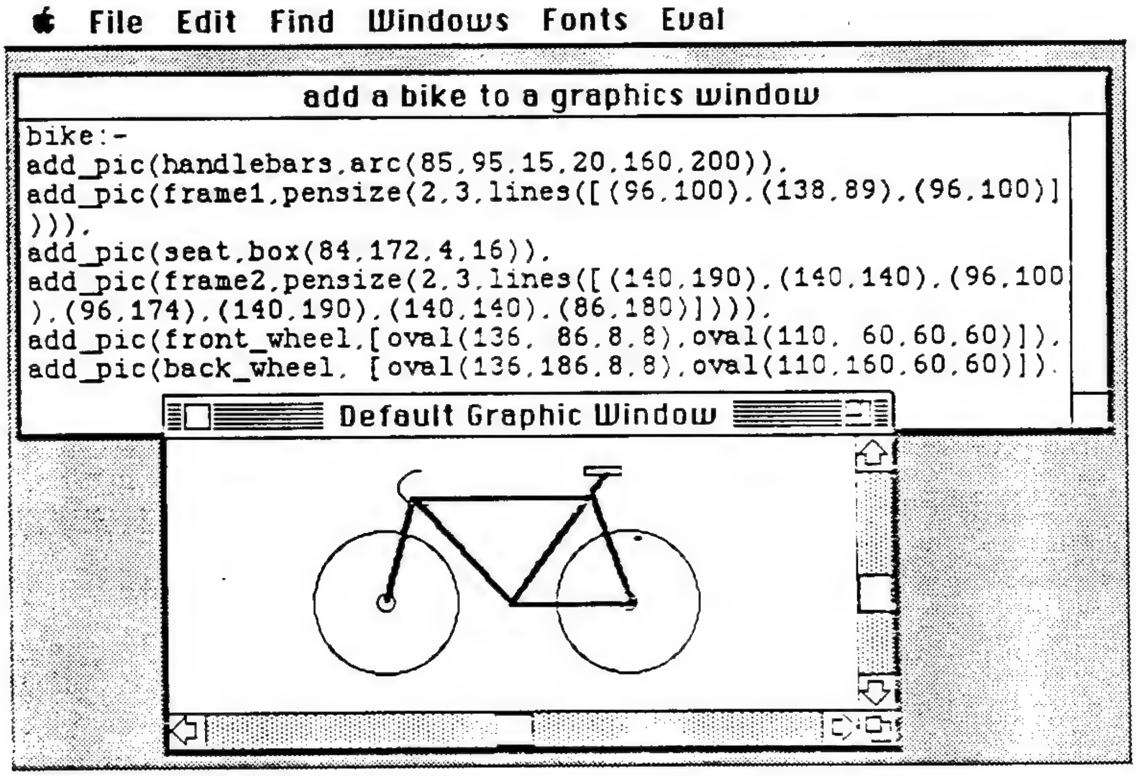
EXAMPLE 1

The `bike` program defined in the program window below will draw a bike in the **Default Graphic Window** as six separate picture entities, because there are 6 calls to `add_pic`. Each picture is a separate logical object. These pictures are

`handlebars, frame1, frame2, back_wheel, front_wheel` and `seat`

Each can selected and manipulated independently from the others.

The bike is drawn with the call `bike`.



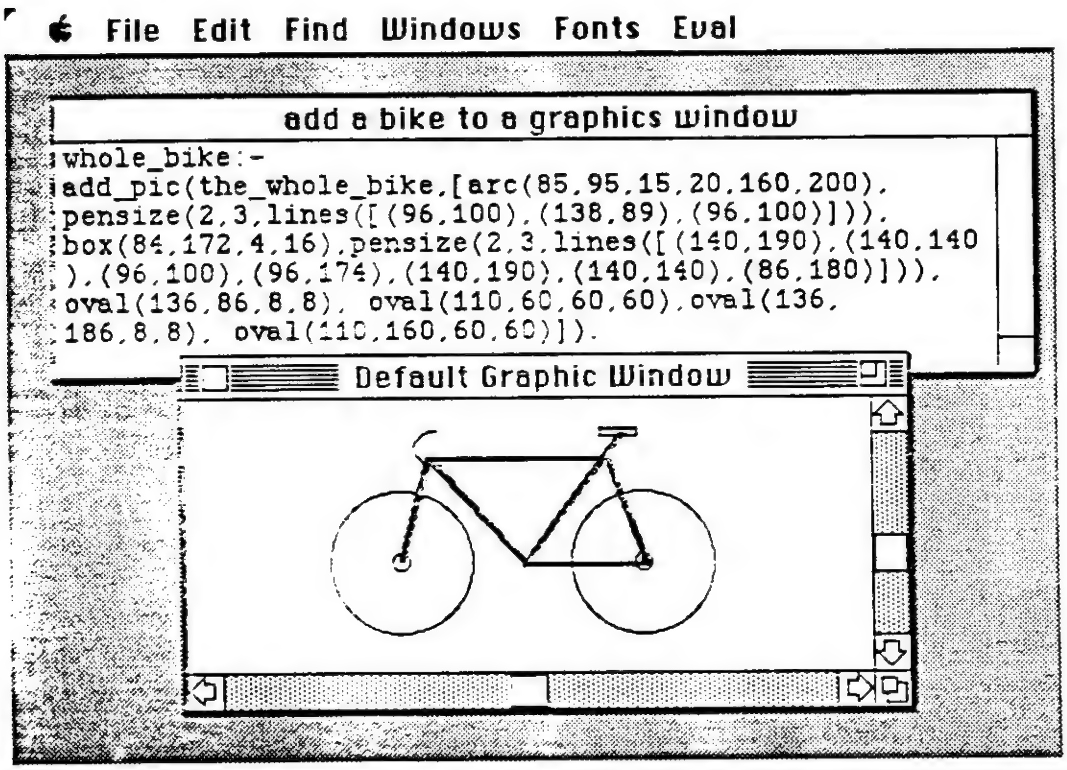
`box, arc, lines,` and `oval` are elementary picture descriptors.

`thin, double, pensize, black, blackpen` are transformation descriptors.

EXAMPLE 2

An alternative definition, using only one call to `add_pic`, draws the same picture as one picture aggregate. Though the bike picture looks the same, the difference is that now the bike is one logical structured picture, and the operations of selection, dragging, and 'get information' will now treat the bike as *one* unit.

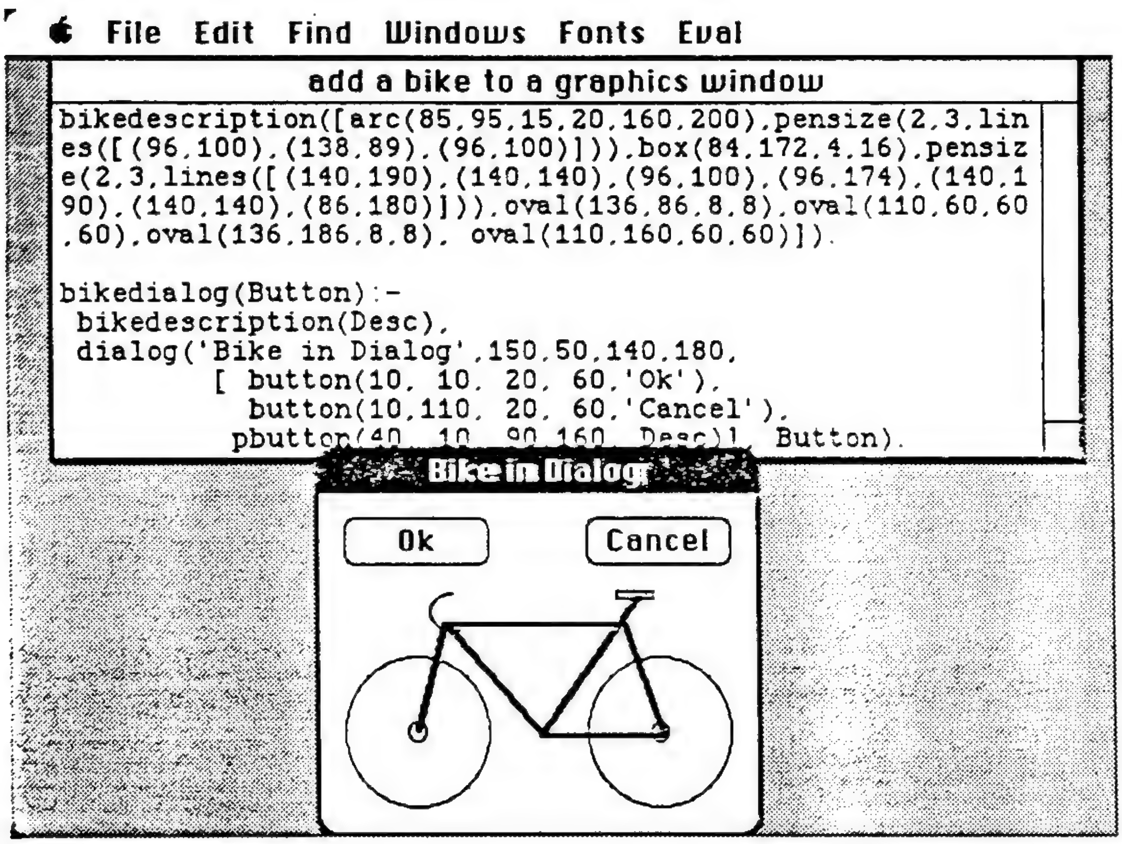
The bike is drawn with the call `whole_bike`.



EXAMPLE 3

The same bike could be displayed in a dialogue by having its graphical description as the field descriptor of a `pbutton` or `pcheck` format item (see the chapter on Advanced Dialogues).

This is shown in the example program below.



The elementary picture descriptors

Elementary picture descriptors describe two types of pictures: hollow and filled.

A *hollow* picture has an outline only, drawn in a particular pen. The default pen pattern is `black`, its default size is `thin`, and its default mode is `paint`. These defaults can be changed by applying a picture transformation. Pictures underneath a hollow picture remain visible; the centre of a hollow object is transparent.

A *filled* picture has both an outline and an interior. This interior is filled using a fill pattern. The default fill pattern is `black`, and can be changed by applying a picture transformation. Pictures underneath a filled picture do not remain visible; the centre of a filled object, even if it is filled with white, is non-transparent.

So while

```
      box(10,20,30,40)
and   white(fillbox(10,20,30,40))
```

describe similar looking objects, they affect the appearance of pictures underneath them in different ways.

Remember that the elementary picture descriptors that follow cannot be called directly; they can only be used as components of picture terms which are passed as arguments to other Graphics primitives such as `add_pic` or `draw_pic`.

24.1 box - a hollow (optionally round cornered) rectangular box

```
box(top,left,depth,width)
box(top,left,depth,width,ovaldepth,ovalwidth)
```

ARGUMENTS

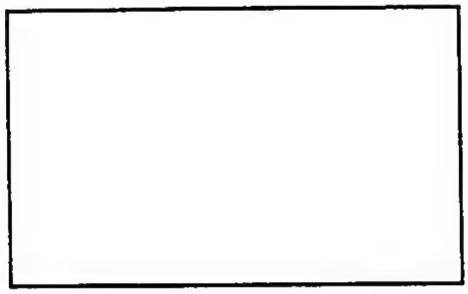
top	: integer, top edge of box
left	: integer, left edge of box
depth	: integer, depth of box
width	: integer, width of box
ovaldepth	: integer, height of oval for rounding of corners
ovalwidth	: integer, width of oval for rounding of corners

USES

1. Four argument use

To describe a hollow rectangular box. By default, the box has a thin black outline, which can be modified using a transformation descriptor.

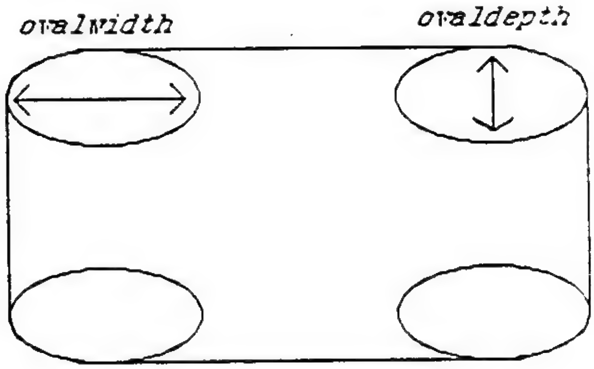
For example, `box(20,20,110,180)` describes:



and `thick(grey(box(20,20,110,180)))` describes the same rectangle but with a thick grey outline.

2. Six argument use

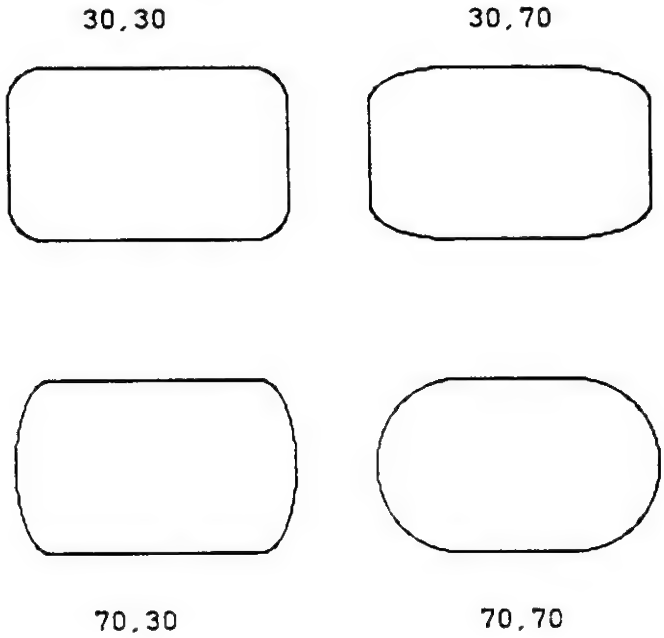
To describe a hollow round cornered rectangular box. *ovaldepth* and *ovalwidth* specify the diameters of curvature for the corners. These arguments specify an oval, which determines the 'roundedness' of the corners. By default, the box has a thin black outline.



(When a rounded corner rectangle is drawn, the ovals inside the corners are not actually drawn.)

EXAMPLES

The following rounded rectangles each have same *depth* and *width* for their rectangular dimensions, but have differing *ovaldepth* and *ovalwidth*.



24.2 `fillbox` - a filled (optionally round cornered) rectangular box

fillbox(*top*, *left*, *depth*, *width*)
fillbox(*top*, *left*, *depth*, *width*, *ovaldepth*, *ovalwidth*)

ARGUMENTS

top	: integer, top edge of box
left	: integer, left edge of box
depth	: integer, depth of box
width	: integer, width of box
ovaldepth	: integer, height of oval for rounding of corners
ovalwidth	: integer,width of oval for rounding of corners

USES

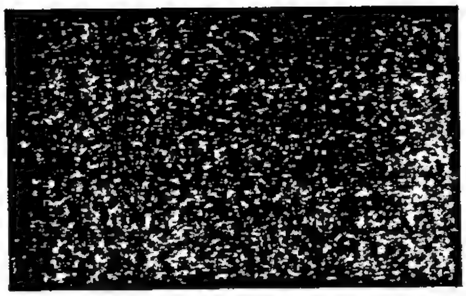
1. Four argument use

To describe a filled rectangular box. By default, the box has a thin black outline and is filled with black, which can be modified using a transformation descriptor.

For example,

```
fillbox(20,20,110,180)
```

describes:



and `check(fillbox(20,20,110,180))` describes the same rectangle but with a checked interior.

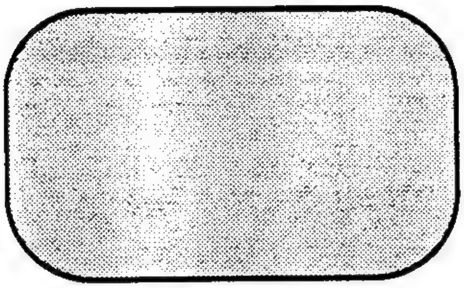
2. Six argument use

To describe a filled round cornered rectangular frame. *ovaldepth* and *ovalwidth* specify the diameters of curvature for the corners, as before. By default, the oval has a thin black outline and is filled with a black pattern. Both can be modified using transformation descriptors.

For example,

```
grey(double(fillbox(20,20,110,180,70,70)))
```

describes:



where `grey` sets the fill pattern to grey, and `double` doubles the size of the pen, giving a thicker black outline.

24.3 **`oval`** - a hollow oval

`oval(`*`top,left,depth,width`*`)`

ARGUMENTS

`top`	: integer, top edge of oval's enclosing box
`left`	: integer, left edge of oval's enclosing box
`depth`	: integer, depth of oval
`width`	: integer, width of oval

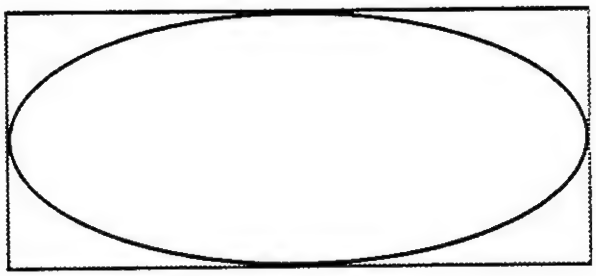
USE

To describe a hollow oval which just fits in the rectangular frame specified by the arguments. If the frame is square then this gives a circle. By default, the oval has a thin black outline.

For example,

```
double(oval(20,20,100,230))
```

describes



The `double` transformation gives the oval a thicker outline.

(The dotted rectangular frame will not be drawn, but here it indicates the frame into which the oval fits.)

24.4 **`filloval`** - a filled oval

`filloval(`*`top,left,depth,width`*`)`

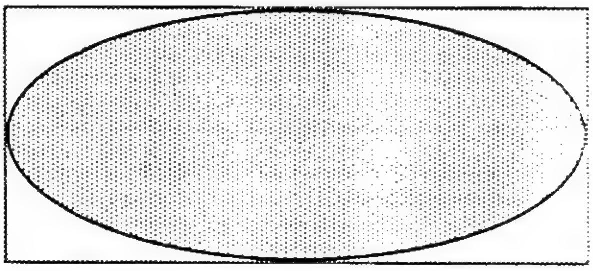
ARGUMENTS

`top`	: integer, top edge of oval's enclosing box
`left`	: integer, left edge of oval's enclosing box
`depth`	: integer, depth of oval
`width`	: integer, width of oval

USE

To describe an oval which just fits in the rectangular frame. If the frame is square then the oval is a circle. By default, the oval has a thin black outline and is filled with black. The defaults can be modified using transformation descriptors.

For example, `grey(filloval(20,20,100,230))` describes



(The dotted rectangular frame will not be drawn, but here it indicates the frame into which the oval fits.)

24.5 **`circle`** - a hollow circle

`circle(`*`ycentre,xcentre,radius`*`)`

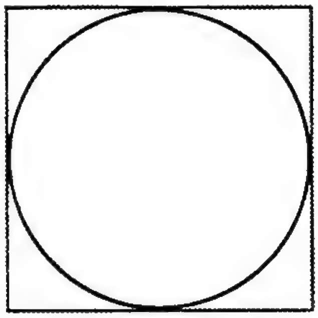
ARGUMENTS

`ycentre`	: integer, vertical centre of circle
`xcentre`	: integer, horizontal centre of circle
`radius`	: integer, radius

USE

To describe a hollow circle centred at *`(ycentre,xcentre)`* and of a given *`radius`*. By default, the circle has a thin black outline.

For example: `circle(75,110,60)` describes:



(The dotted rectangular frame will not be drawn, but indicates the frame into which the circle fits.)

24.6 `fillcircle` - a filled circle

`fillcircle(`*`ycentre`*`,`*`xcentre`*`,`*`radius`*`)`

ARGUMENTS

ycentre	: integer, vertical centre of circle
xcentre	: integer, horizontal centre of circle
radius	: integer, radius

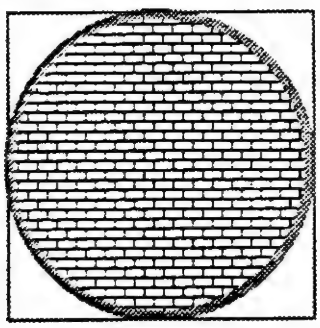
USE

To describe a circle centred at (*ycentre*, *xcentre*) and of a given *radius*. By default, the circle has a thin black outline and is filled with black. The defaults can be modified using transformation descriptors.

For example,

```
greypen(pensize(3,3,brick(fillcircle(75,110,60))))
```

describes



where `brick` sets the fill pattern to a brick like pattern, `greypen` sets the pen pattern to grey, and `pensize` sets the size of the pen.

24.7 **`square`** - a hollow square frame

`square(`*`ycentre,xcentre,size`*`)`

ARGUMENTS

`ycentre`	: integer, vertical centre of square
`xcentre`	: integer, horizontal centre of square
`size`	: integer, size of sides of square

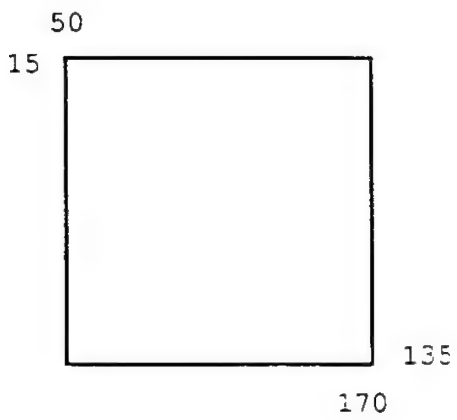
USE

To describe a hollow square centred at (*`ycentre,xcentre`*) and with sides of given *`size`*. By default, the square has a thin black outline.

For example,

```
square(75,110,120)
```

describes:



24.8 **`fillsquare`** - a filled square

`fillsquare`(*`ycentre`*, *`xcentre`*, *`size`*)

ARGUMENTS

`ycentre`	: integer, vertical centre of square
*`xcentre`*E	: integer, horizontal centre of square
`size`	: integer, size of sides of square

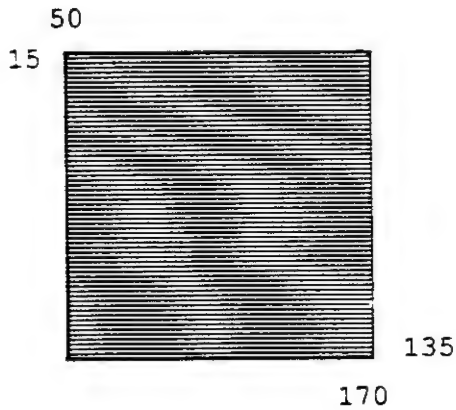
USE

To describe a square centred at (*`ycentre`*, *`xcentre`*) and with sides of a given *`size`*. By default, the square has a thin black outline and is filled with black.

For example,

```
horiz(fillsquare(75,110,120))
```

describes:



where `horiz` sets the fill pattern to a horizontal pattern .

24.9 **arc - a hollow arc**

`arc(top, left, depth, width, startangle, arcangle)`

ARGUMENTS

top	: integer, top edge of oval's enclosing box
left	: integer, left edge of oval's enclosing box
depth	: integer, depth of oval
width	: integer, width of oval
startangle	: integer, beginning of arc
arcangle	: integer, degrees clockwise

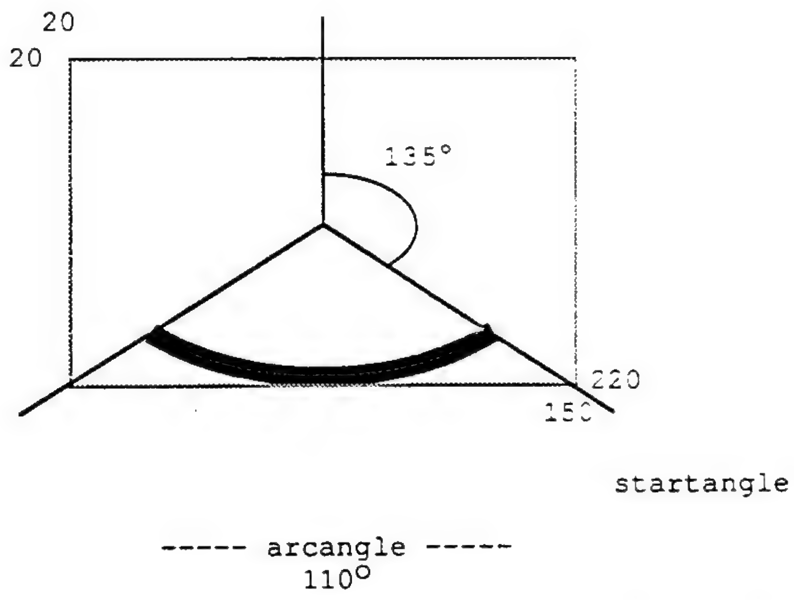
USE

To describe an arc as a pie-segment of the oval defined by *top, left, depth, width*.

The size of this segment is determined by the *startangle, arcangle* arguments, where *startangle* indicates the angle at which the arc begins and *arcangle* defines the angular extent of the arc. Angles can be given in positive or negative degrees, where a positive angle goes clockwise, and a negative angle goes counter-clockwise.

A *startangle* of zero degrees is at 12 o'clock high, 90 (or -270) at 3 o'clock, 180 (or -180) at 6 o'clock and 270 (or -90) is at 9 o'clock.
By default, the arc has a thin black outline.

Example, `thick(arc(20,20,130,200,135,110))` describes:



where the dotted rectangle is an indication of the rectangle into which the complete oval would fit, and the `thick` transformer sets the pen to a size of 8*8 pixels.

24.10 wedge - a filled arc

wedge(*top,left,depth,width,startangle,arcangle*)

ARGUMENTS

top	: integer, top edge of oval's enclosing box
left	: integer, left edge of oval's enclosing box
depth	: integer, depth of oval
width	: integer, width of oval
startangle	: integer, beginning of wedge
arcangle	: integer, degrees clockwise

USE

To describe a filled arc as a pie-segment of the oval defined by *top, left, depth, width.*

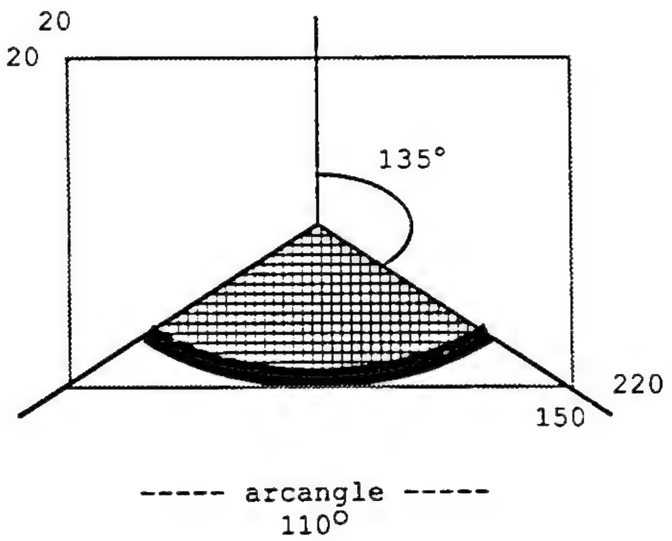
The size of this segment is determined by the *startangle, arcangle* arguments, where *startangle* indicates the angle at which the arc begins and *arcangle* defines the angular extent of the arc. Angles can be given in positive or negative degrees; where a positive angle goes clockwise, and a negative angle goes counter-clockwise.

A *startangle* of zero degrees is at 12 o'clock high, 90 (or -270) at 3 o'clock, 180 (or -180) at 6 o'clock and 270 (or -90) is at 9 o'clock.
By default, the wedge has a thin black outline and is filled with black.

For example,

```
thick(boxed(wedge(20,20,130,200,135,110)))
```

describes



where boxed sets the fill pattern to be a box-like pattern and the dotted rectangle is an indication of the rectangle into which the oval segment fits.

Note: The radii lines of the segment are not drawn.

24.11 `lines` - connected lines

`lines(`*`listofpoints`*`)`

ARGUMENT

`listofpoints` : list of points

USE

To describe a sequence of joined lines between the points in the *`listofpoints`*. By default, the lines will be drawn in a thin black line.

The list of points is of the form

```
[(y0,x0),(y1,x1), ...]
```

or

```
[[y0,x0],[y1,x1], ...]
```

where `(y,x)` is a point represented by a comma pair
and `[y,x]` is a point represented by a two element list

For example,

```
pensize(1,2,lines([(-10,50),(-20,90),(-10,90),(2,62),
                   (15,90),(25,90),(15,50),(-10,50)]))
```

describes

The `pensize` transformer sets the size of the pen to the given *`depth`* and *`width`* arguments.

24.12 `poly` - an enclosed object of connected lines

`poly(`*`listofpoints`*`)`

ARGUMENT

`listofpoints` : list of point pairs

USE

To describe a closed sequence of joined lines between the points in the *`listofpoints`*. By default, the lines will be drawn in a thin black line.

The list of points is of the form

`[(y0,x0),(y1,x1), ...]`

or

`[[y0,x0],[y1,x1], ...]`

where `(y,x)` is a point represented by a comma pair
and `[y,x]` is a point represented by a two element list

The set of lines will be automatically closed by a line from the last point back to the first point.
The main difference between `lines` and `poly` is that polygons can be inverted.

24.13 `fillpoly` - a filled enclosed object of connected lines

`fillpoly(`*`listofpoints`*`)`

ARGUMENT

`listofpoints` : list of points

USE

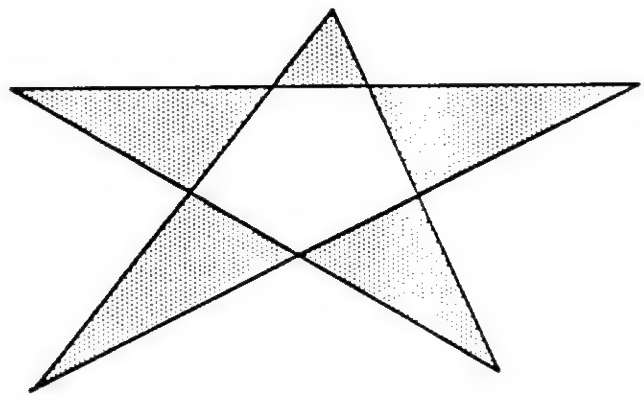
The same as `poly` except that the area enclosed by the set of closed lines is filled with the fill pattern, which by default is black.

The system will add a line from the last point back to the first point.

EXAMPLE 1

```
lgrey(fillpoly([(119,24),(120,22),(1,265),
                                (1,16),(114,208),(-23,144)]))
```

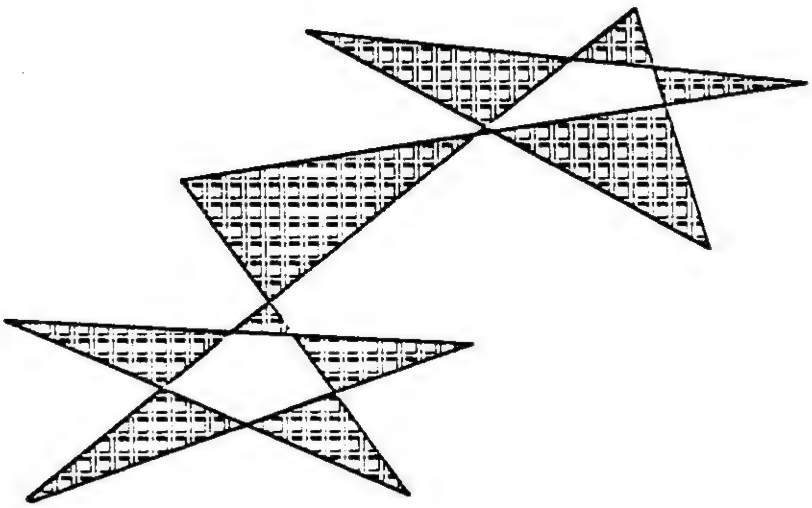
describes



EXAMPLE 2

```
alpha(fillpoly([(155,-16),(94,160),(84,-26),(153,134),
                (29,45),(-9,294),(-30,96),(56,254),
                (-39,226),(154,-18)]))
```

describes

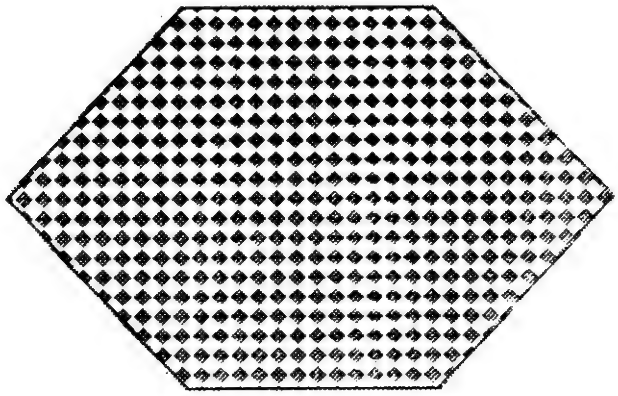


The Macintosh fills polygons by counting each line in the list of lines defining the polygon as a boundary line. Then, as it traverses each boundary line it toggles the filler on/off. So, in these two examples above, the polygons have lines that cross each other and they therefore have some unfilled interior areas.

EXAMPLE 3

```
greypen(diamonds(fillpoly([(50,-20),(125,50),(125,150),
                    (50,220),(-25,150),(-25,50)])))
```

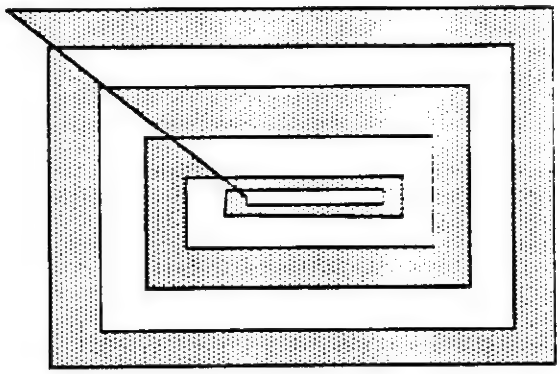
describes



EXAMPLE 4

```
lgrey(fillpoly([(111,310),(114,310),(114,364),(108,364),(108,302),
      (118,301),(119,372),(103,372),(103,286),(131,286),(131,384),
      (87,384),(87,270),(147,270),(147,399),(67,399),(67,252),
      (164,252),(164,416),(51,416),(51,232),(178,232),
      (178,432),(36,432), (36,216)])
```

describes



24.14 `text` - a text display

`text(font,size,face,ypos,xpos,text)`

ARGUMENTS

fontname	: atom, name of font
fontsize	: integer, point size
fontface	: integer, face details
ypos	: integer, vertical position of bottom of first character
xpos	: integer, horizontal position of first character
text	: atom, text to enter

USE

To describe a text display in a particular font, face and size and with the bottom left end of the first character at `(ypos,xpos)`.

The *fontname* is `'Times'`, `'Courier'`, `'Geneva'`, `'Helvetica'` or any other font supported by your system disk.

The *fontsize* is 10, 12, 14, 18, 24

The *fontface* can be

0	for	Normal
1	for	**Bold**
2	for	*Italic*
4	for	Underline
8	for	Outline
16	for	Shadow

This can be combined by addition, resulting in a composite type face

e.g 1 + 2 + 16 = 19 ***Italic + Bold + Shadow***

24.15 `textbox` - a text display in a box

`textbox(`*`font,size,face,t,l,d,w,just,text`*`)`

ARGUMENTS

font	: atom, name of font
size	: integer, point size
face	: integer, face details
t	: integer, top edge of box
l	: integer, left edge of box
d	: integer, depth of box
w	: integer, width of box
just	: integer, text justification
text	: atom, text to display

USES

To describe a text display in a particular *font*, *face* and *size*, placed in a box, with the first letter positioned in the top left hand corner of the box, and carriage returns automatically inserted as required (see the description of `text` for details of *font*, *face* and *size*).
The box outline will not be drawn.

The *just* argument determines how the text is justified within the box. It may be

(for left justified text
-1 for right justified text
1 for centered text

As an example:

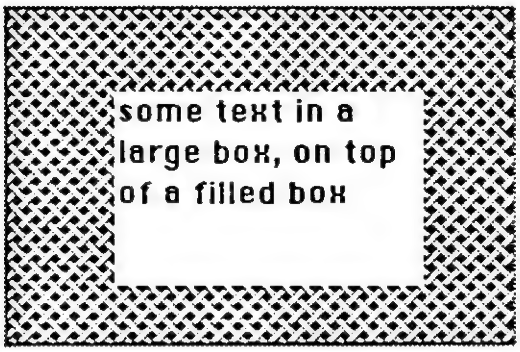
```
textbox('Chicago',12,0,-18,7,121,192,0,
'Here is an example of some text in a box. It is Chicago,
normal, and size 12.~MYou can insert carriage returns, and type
past the end of the box, in which case you will only see the text
that is in the box ')
```

describes:

**Here is an example of some
text in a box. It is Chicago,
normal, and size 12.
You can insert carriage
returns, and type past the
end of the box, in which
case you will only see the
text that is in the box**

The font is Chicago 12 point, and the *just* argument is 0 so the text is displayed with left justification.

The picture below is of a text box drawn on top of a filled box. As can be seen, the empty parts of the text box appear white, and obscure any picture underneath.



24.16 `icon` - an icon defined by two 'hexadecimal strings'

```
icon(top,left,depth,width,hex1,hex2)
icon(top,left,depth,width,icon_details)
```

ARGUMENTS

top	: integer, top edge of icon
left	: integer, left edge of icon
depth	: integer, depth of icon
width	: integer, width of icon
hex1	: atom, comprising 128 hexadecimal characters
hex2	: atom, comprising 128 hexadecimal characters
icon_details	: resource number of icon, or term identifying resource icon

USES

1. Six argument use
To describe an icon represented as two atoms of 128 hexadecimal characters, where in the four-bit binary representation of each hex character, 0 represents a white pixel and 1 represents a black pixel. An icon is a grid of 32 x 32 pixels. One row of the icon is represented by eight hexadecimal characters. The two atoms of 128 four-bit characters represent the 1028 individual pixels that are needed to define an icon. The first atom encodes, row-wise, the top half of the icon, and the second the bottom half.

Examples of rows:

'00000000'	completely white
'33333333'	grey
'FFFFFFFF'	completely black

ICON EXAMPLE

Suppose that

```
Hex1  =
'0010200000186000071 4A38002D32D0001201200008004000040080000 7FF80000
201000002010000 01FE0000008400000084000000840000008400000084000',
```

and

```
Hex2  =
'00084000000840000008400000084000000840000 03FF0000020100000020100000
FFFC0001800600030003000200010007FFFF8007FFFF80000000000000000000'.
```

The description

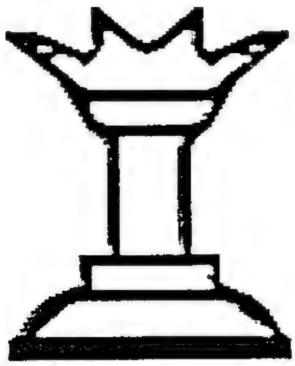
```
icon(0,0,32,32,Hex1,Hex2)
```

describes the icon

and the description

```
icon(0,0,200,200,Hex1,Hex2)
```

describes



The 32x32 bit image of the icon is scaled and shifted to fit into the rectangle enclosed by `(top,left,depth,width)`. To guard against possible distortion of the bit image, it is advisable to ensure that this rectangle's height and width are both multiples of 32.

NOTE: The reason why we need two atoms of hex digits rather than one, is that the maximum length of an atom is 255 characters. However, 256 digits are needed to fully encode the 32x32 bit image.

2. <u>Five argument use</u>
To describe an icon which exists in a resource file.

`icon_details` indicates a (resource) icon on disk. It is a term of the form:

	(i) *`resourceid`*
or	(ii) `resource(`*`name`*`)`
or	(iii) `resource(`*`name, file`*`)`
or	(iv) `resource(`*`name, file, volume`*`)`
or	(v) `resource(`*`number`*`)`
or	(vi) `resource(`*`number, file`*`)`
or	(vii) `resource(`*`number, file, volume`*`)`

The icon is scaled to fit the rectangle defined by *`top`*, *`left`*, *`depth`*, *`width`*.

The following icons from MacPROLOG 2.0 and the Macintosh System (Version 5.3) are always available, and have the following *`resourceids`* :

1000 1001 1002

1003 1007 0 1 2

1008 1009

You can get a complete list of all the icons currently available in the system using the `res_items` primitive (see the chapter on Resources).

24.17 `picture` - a QuickDraw picture

`picture(top,left,depth,width,picture_details)`

ARGUMENTS

top	: integer, top edge of picture
left	: integer, left edge of picture
depth	: integer, depth of picture
width	: integer, width of picture
picture_details	: term identifying resource picture

USE

To describe a picture imported from a resource file. All imported pictures can be repositioned and rescaled by changing the rectangle into which the picture is to be drawn, and the picture will be expanded or shrunk as needed.

The *picture_details* argument may be as follows.

(a) A (resource) picture on disk. It is a term of the form:

(i) `resource(name)`
or (ii) `resource(name,file)`
or (iii) `resource(name,file,volume)`
or (iv) `resource(number)`
or (v) `resource(number,file)`
or (vi) `resource(number,file,volume)`

Resource pictures can be loaded from resource files and then drawn as QuickDraw pictures. Resource pictures are stored in the resource forks of files, and always have associated with them an identifying number, and often an identifying name. They can be referenced by either attribute.

Before drawing a resource, its resource file must be opened.

In the two cases where the file name and volume are given in the *picture_details*, the file is automatically opened, if found.

In the two cases where the file name but not the volume is given in the *picture_details*, the file is automatically opened and assumed to be on the default volume.

In the two cases where the file name is not given, the onus is on the programmer to ensure that the resource file is currently open (e.g. by using the `res_open` primitive; see chapter on Resources).

Displaying a QuickDraw equivalent of some complicated GDL picture is much faster than displaying the description. If an application is going to display some complex GDL picture frequently, then it is more efficient to construct a QuickDraw picture from the GDL description, and then use the `picture` primitive to display it. The `save_pic` primitive allows you to save a GDL description as a resource picture.

· The following resource picture is available to you as it originates from the Macintosh System Folder, and can be described by

```
    picture(10,10,50,150,resource(-15968))
or  picture(10,10,50,150,resource(-15968,system))
```

Although this picture looks like several pictures, it is treated as one object. To display just part of it would require aggregating it with some filled white boxes to hide those parts of the picture not required.

(b) A clipboard picture. `picture_details` is a term of the form

```
clipboard('number')
```

If you **Cut** or **Copy** a picture into the Clipboard using the **Quickdraw format**, and then **Paste** it into a graphics window, the description of the picture added to the data base of the graphics window will be a **picture** term, where the `picture_details` argument is a term of the form

```
clipboard('number')
```

and *number* is automatically generated by the system. This picture is added by the system to the graphics window into which the **Paste** took place, using `add_pic`, and is then treated as one logical unit. A name for the picture is automatically generated.

Transformation Descriptors

A transformation descriptor does not directly specify a visible object, rather it **modifies** the appearance of any picture defined within its scope. Any picture descriptions which are not enclosed by the descriptor are unaffected.

Transformation descriptors can be used in combination, e.g.

```
check(fillbox(10,20,130,140))
```

describes a box filled with a checked pattern,

```
greypen(check(fillbox(10,20,130,140)))
```

describes a box filled with a checked pattern, and drawn with a grey outline, and

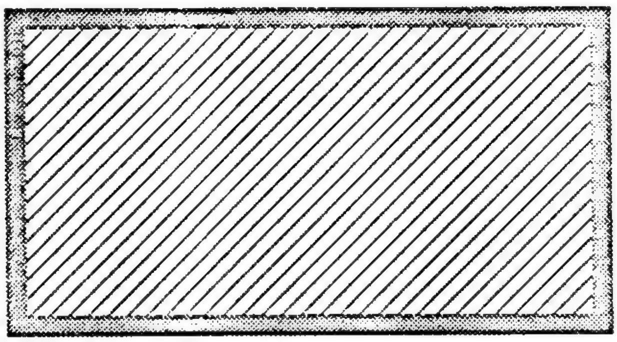
```
thick(greypen(check(fillbox(10,20,130,140))))
```

describes the same box but drawn with a thick grey pen.

In the case of conflicting descriptors, the **outermost** descriptor has priority, e.g.

```
diag(thick(greypen(check(fillbox(10,20,130,140)))))
```

represents a box as drawn below.



where the descriptor `diag` has overridden the descriptor `check` in determining the fill pattern.

24.18 **trans** - translate a picture

`trans(down,across,picture)`

ARGUMENTS

down	: integer, vertical translation
across	: integer, horizontal translation
picture	: a picture description

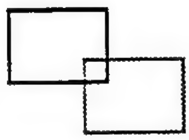
USE

To translate a `picture` by `down` vertically and `across` horizontally. This is an inexpensive mapping since only the logical origin of the picture is affected. Pictures can be translated in both positive and negative directions.

For example,

```
[box(10,20,30,40),greypen(trans(20,30(box(10,20,30,40))))]
```

describes the following picture:



24.19 **scale** - scale a picture

`scale(vertscale,horizscale,v_org,h_org,picture)`
`scale(vertscale,horizscale,picture)`

ARGUMENTS

vertscale	: number, vertical scaling factor
horizscale	: number, horizontal scaling factor
v_org	: integer, vertical origin of scale
h_org	: integer, horizontal origin of scale
picture	: a picture description

USE

To scale a `picture` by a factor of `vertscale` vertically and `horizscale` horizontally. The scaling is performed relative to the given origin of scale `(v_org,h_org)`. In the three argument form, the scaling is done about (0,0).

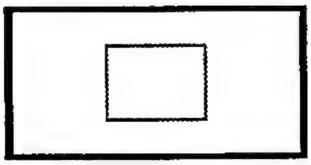
By scaling about the local centre of a picture, one can expand (or shrink) the picture whilst its centre stays fixed. You can find a picture's centre using the `pic_centre` primitive (see the chapter on Picture Manipulation). Scaling about any other point will have the effect of moving the picture's centre.

Note that scaling factors less than 1 will shrink a picture. The scaling also affects the pen size, and if this causes the pen's depth or width to be less than 1, the pen becomes invisible.

EXAMPLE 1

The centre of both boxes below is (25,40). By scaling the second box about this origin, we have scaled in place.

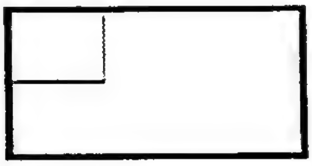
```
[greypen(box(10,20,30,40)),scale(2,3,25,40,box(10,20,30,40))]
```



EXAMPLE 2

Both boxes below have their top, left hand corners at (10,20). By scaling the second box about this origin, we have 'stretched' it and kept the top left hand corner fixed.

```
[greypen(box(10,20,30,40)),scale(2,3,10,20,box(10,20,30,40))]
```



EXAMPLE 3

The imported picture displayed in the first **A graphic window** on the next page is described by

```
picture(150,115,180,346,resource(456096,mypictures))
```

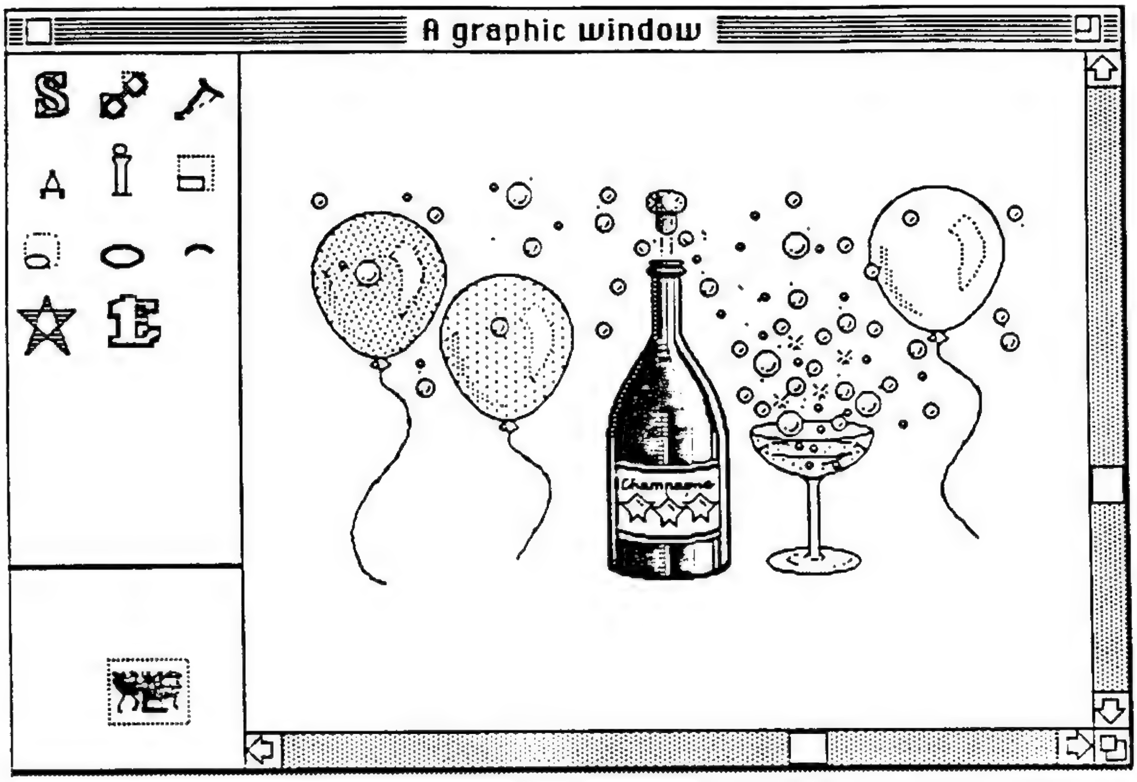
The vertical coordinate of its centre is (150 + 180/2), and the horizontal coordinate of its centre is (115 + 346/2) i.e. at `(240,288)`.
(The picture is stored in a resource file called `mypictures`).

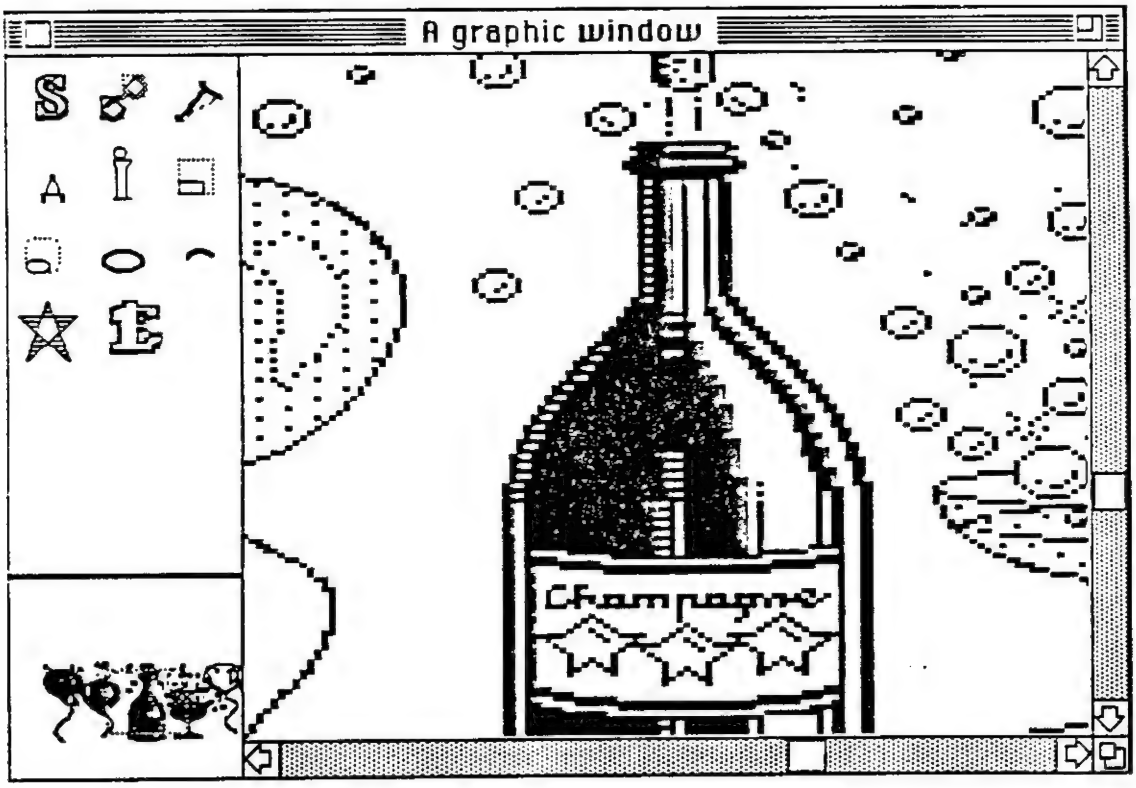
Introducing the `scale` descriptor and changing its description to

```
scale(2,3,240,288,
    picture(150,115,180,346,resource(456096,mypictures)))
```

enables us to obtain the second picture shown below.

Notice that the vertical scaling factor is 2 whilst the horizontal scaling factor is 3, and therefore the picture has been 'stretched' more in the horizontal direction than in the vertical direction, as can be seen by a close examination of the bubbles!


A graphic window
Champagne


A graphic window
Champagne

The Pen

The outline of all elementary pictures is drawn in the current pen. This pen has three attributes, `pensize`, `penpattern` and `penmode`, all of which can be transformed using the built-in system descriptors or user-defined forms.

The Pen's size

Pen size transformers fall into two categories: <u>absolute</u> and <u>relative</u>.

Absolute transformers reset the pen to a certain fixed size regardless of what its present size is. This means that they ignore other pen size descriptors which fall within their scope, in a similar way to `penpattern` and `fillpattern` transformers.

For example:

```
pensize(2,3,double(thick(box(10,20,30,40))))
```

represents a box drawn in a pen whose dimensions are 2 pixels deep, and 3 pixels wide. The fact that `thick` occurs inside `pensize` means it is ignored. This enables you to prefix any picture description with an absolute pen size and ensure it is drawn in the pen you want, regardless of its description.

Relative transformers change the pen relative to its present description.

For example, if the current pen size is 1 x 1 pixels,

```
thicker(2,3,box(10,20,30,40))
```

represents a box drawn in a pen whose dimensions are 3 pixels deep, and 4 pixels wide, and

```
thicker(2,3,thicker(2,3,box(10,20,30,40)))
```

represents a box drawn in a pen whose dimensions are 5 pixels deep, and 7 pixels wide.

NOTE

If a pen size transformer causes the pen's size to become less than 1, the pen will become invisible.

Absolute pen size transformers

24.20 `thin` - set pen to 1x1 pixels in size

`thin(picture)`

ARGUMENT

picture : a picture description

USE

To set the size of the pen to 1 x 1 pixels. This is actually the default pen size.

24.21 `thick` - set pen to 8x8 pixels in size

`thick(picture)`

ARGUMENT

picture : a picture description

USE

To set the size of the pen to 8 x 8 pixels.

24.22 `nilpen` - set pen to 0x0 pixels in size

`nilpen(picture)`

ARGUMENT

picture : a picture description

USE

To set the size of the pen to 0 x 0 pixels, i.e. invisible. This means that filled objects have no outline, and hollow objects are invisible. The combination of

```
nilpen(box(....))
```

is sometimes useful for 'framing' pictures, i.e. giving them a fixed but invisible outline that might affect their behaviour with graphic tools.

24.23 `pensize` - set pen to user defined size

`pensize(`*`depth, width, picture`*`)`

ARGUMENTS

`depth`	: integer, depth of pen in pixels
`width`	: integer, width of pen in pixels
`picture`	: a picture description

USE

To set the size of the pen to *`depth`* x *`width`* pixels.

Relative Pen Size Transformers

24.24 `thinner` - decrease the size of the pen

`thinner(`*`depth, width, picture`*`)`

ARGUMENTS

`depth`	: integer, depth decrement of pen in pixels
`width`	: integer, width decrement of pen in pixels
`picture`	: a picture description

USE

To decrease the size of the pen by the given *`depth`* and *`width`*..

24.25 `thicker` - increase the size of the pen

`thicker(`*`depth, width, picture`*`)`

ARGUMENTS

`depth`	: integer, depth increment of pen in pixels
`width`	: integer, width increment of pen in pixels
`picture`	: a picture description

USE

To increase the size of the pen by the given *`depth`* and *`width`*.

24.26 **`double`** - double the size of the pen

`double(`*`picture`*`)`

ARGUMENT

`picture`	: a picture description

USE

To double the pen's current depth and width.

24.27 **`triple`** - triple the pen's size

`triple(`*`picture`*`)`

ARGUMENT

`picture`	: a picture description

USE

To triple the pen's current depth and width.

24.28 **`penscale`** - scale the pen

`penscale(`*`vertscale,horizscale,picture`*`)`

ARGUMENT

`vertscale`	: number, vertical scale factor
`horizscale`	: number, horizontal scale factor
`picture`	: a picture description

USE

To scale the size of the pen by the *`vertscale`* and *`horizscale`* factors, i.e. the pen's depth is multiplied by *`vertscale`*, and the pen's width is multiplied by *`horizscale`*.

Note that scale factors less than 1 will shrink the pen size, and if the pen size becomes less than 1 then the pen becomes invisible.

The pen's mode

24.29 **penmode - set pen mode**

penmode(*switch,picture*)

ARGUMENTS

switch	: atom, a defined mode
picture	: a picture description

USE

The penmode determines the way the outline of pictures is drawn on the screen. This mode specifies one of eight boolean operations

paint, or, xor, erase, npaint, nor, nxor, nerase

that QuickDraw is to perform before displaying a new pixel.

There are four basic operations -- paint, or, xor, and erase.

The paint operation simply replaces the pixels underneath the pen with the pixels in the source. i.e. it paints pixels onto the screen without regard for what is already there, obscuring any images below. This is the default mode.

The or, xor and erase modes will all draw any white pixels as 'transparent'. i.e. drawing in white in any of these three modes has no visible effect on the screen.
However, when a black pixel is to be drawn, its actual appearance on the screen depends on the pixel already there.
The modes display a black pixel as follows:

or	black
xor	black if the existing screen pixel is white, white if the existing screen pixel is black
erase	white

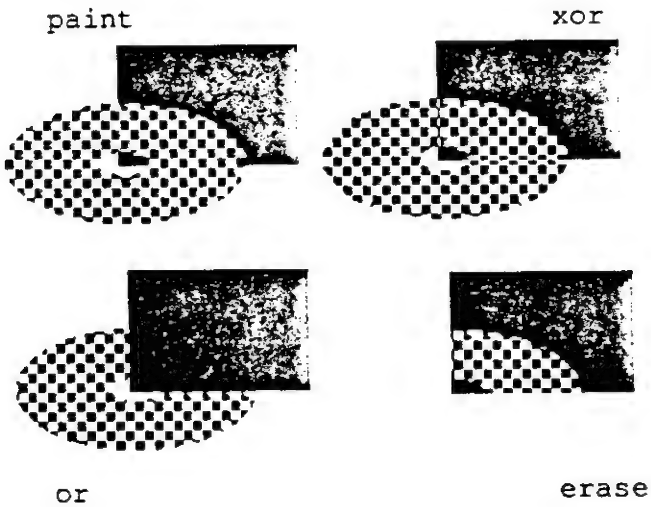
Each of the basic operations has a variant in which every pixel in the pen is inverted before the operation is performed (i.e. npaint, nor, nxor, and nerase), giving eight operations in all.

EXAMPLE 1

In this example, a hollow `oval` has been drawn over a black filled box using a very large pen (described by `pensize(20,40,...))` , and a 'chunky' checkered pattern (described using the `penpattern` descriptor that is explained later in this chapter, with an argument of `'0F0F0F0FF0F0F0F0')`.

The four pictures display the different effects of the four positive pen modes, `paint`, `or`, `xor` and `erase`.

The negated modes are not displayed as they would result in similar images (this pen pattern is symmetrical).



EXAMPLE 2

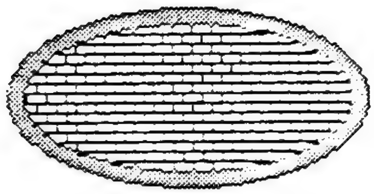
The following eight pictures show how the modes affect the visual appearance of the pen when applied to filled objects. This is because when drawing filled objects, the pen pattern is drawn over the edge of the filled interior.

Each picture below is described by

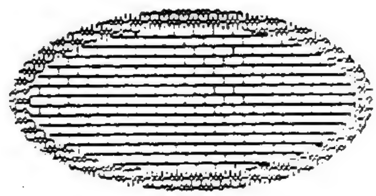
```
greypen(penmode(Mode,thick(brick(filloval(0,0,75,150)))))
```

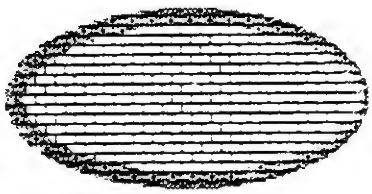
and each picture is labelled by the `Mode` used.

`paint`

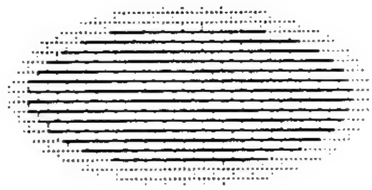


`xor`



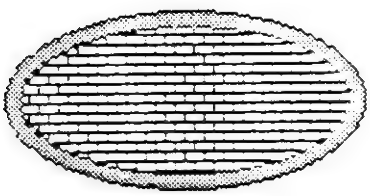


`or`

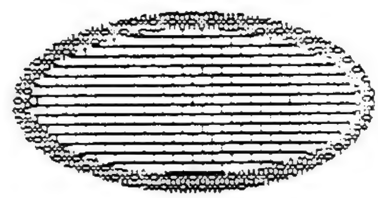


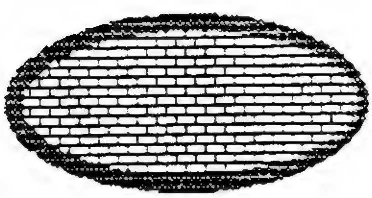
`erase`

`npaint`

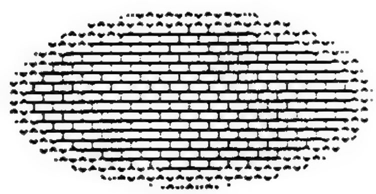


`nxor`





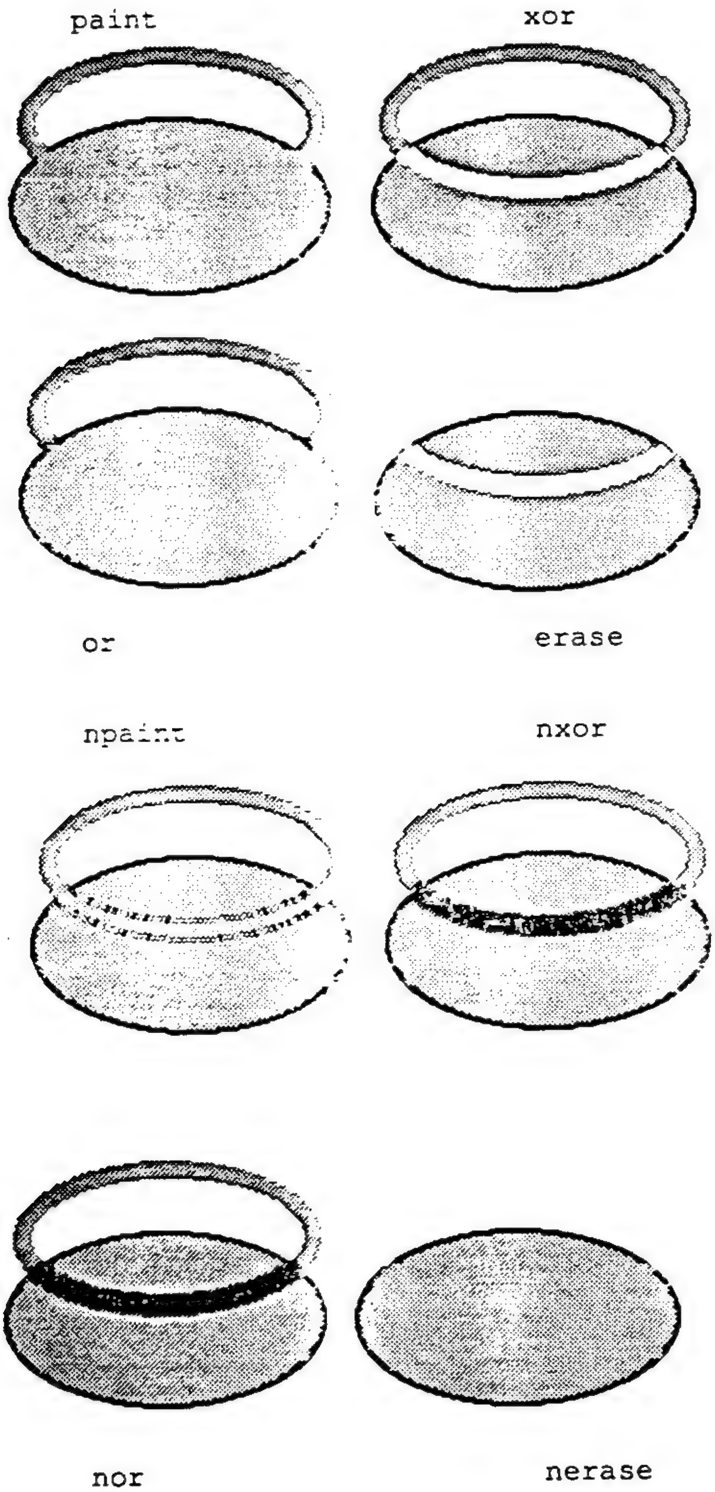
`nor`



`nerase`

EXAMPLE 3

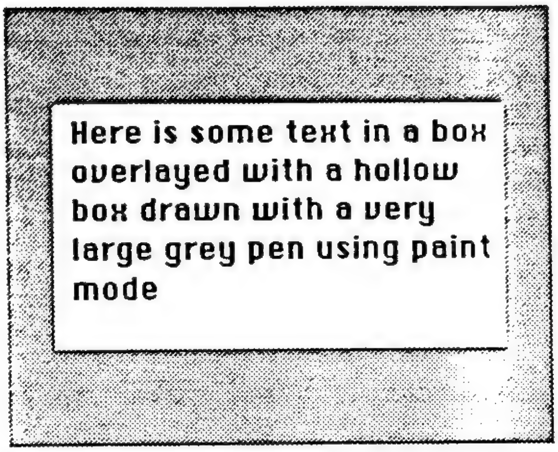
In this example we show how the different pen modes affect the display when a `thick(greypen(oval))` is drawn over a `grey(oval)`

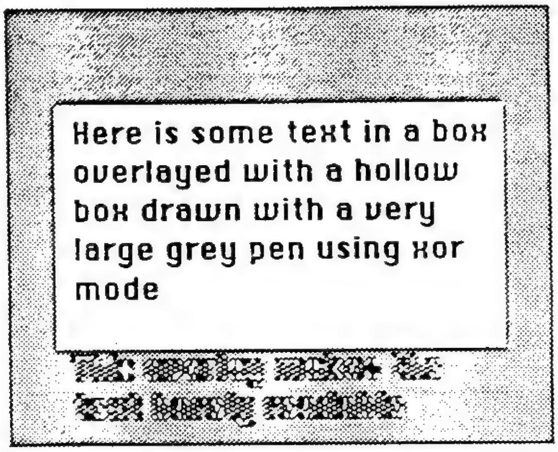


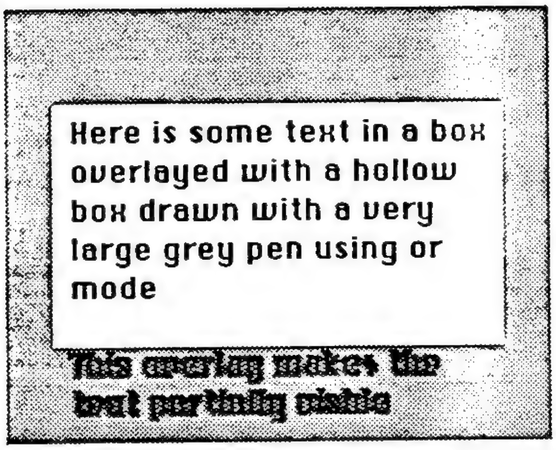
EXAMPLE 4

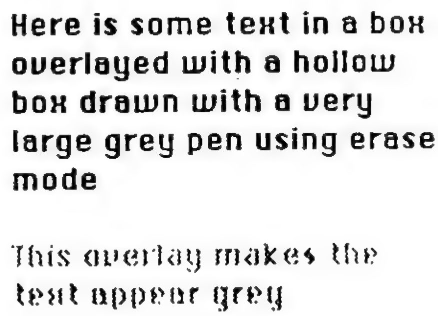
We now look at how combining penmodes with text displays can result in different images. We only show the positive modes, as the negated modes for `greypen` are similar to the positive modes, and for `blackpen` only `npaint` has any effect and that is the same as `erase`.

In these first four pictures the text has been overlayed by a hollow box drawn with a large grey pen.

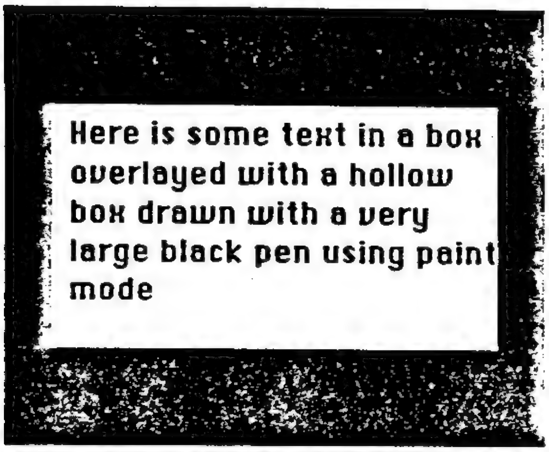


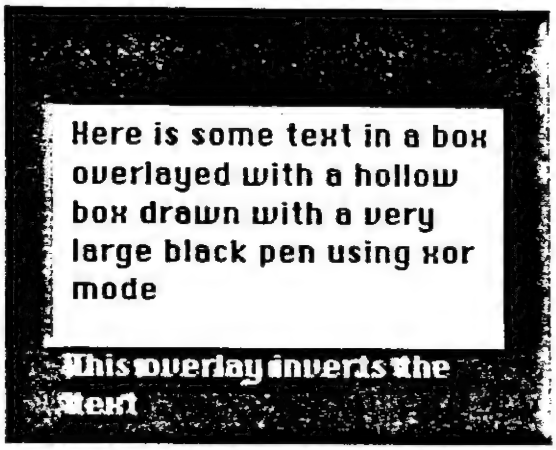


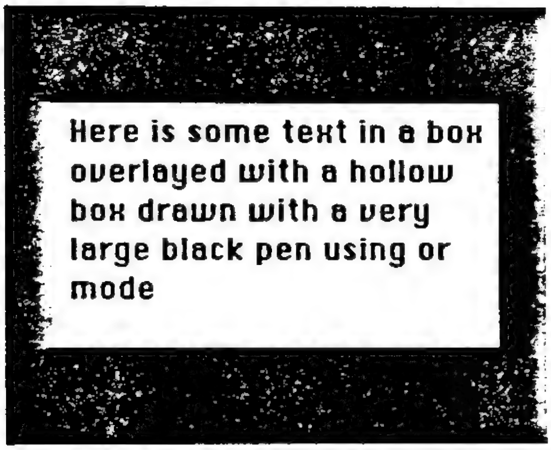


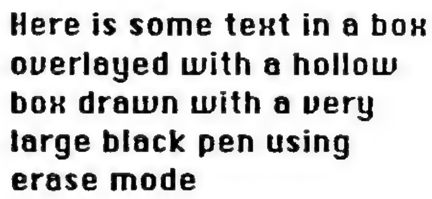


In these four pictures the text has been overlayed by a hollow box drawn with a large black pen.



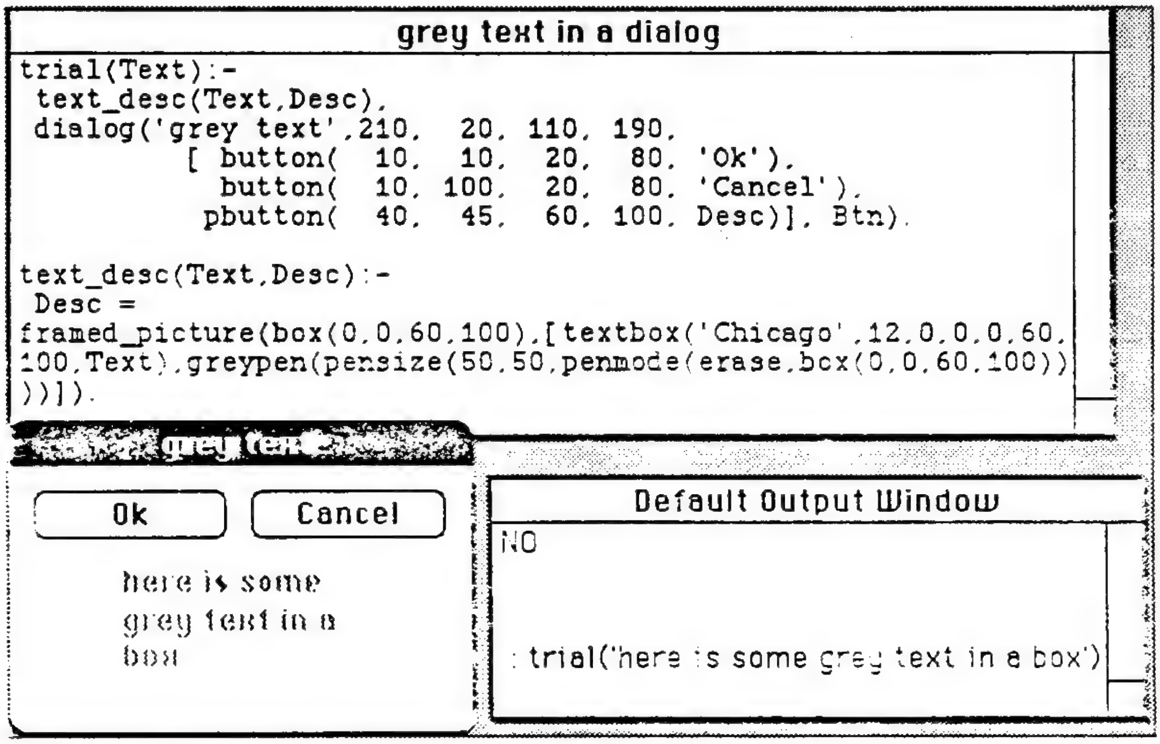






NOTES

(a) The idea of 'grey' text is often very useful in dialogues. This can be achieved by having a disabled `pbutton` or `pcheck` whose picture description contains some text and an overlaying grey box whose penmode is set to `erase`, as in the following example.



In this example we have used the term `framed_picture(Rectangle, Description)` as the picture description. This is a picture description extension, as explained in greater detail at the end of the next chapter on Picture Manipulation.

(As noted previously, when an aggregate picture contains overlapping sub-pictures, QuickDraw does not always return the correct 'frame' for the picture. To overcome this problem we explicitly set the *Rectangle*. We make it the same depth and width as the rectangle of the `pbutton` in the dialogue to ensure that there is no visual distortion in the appearance of the text.)

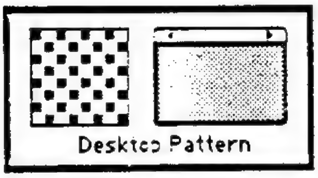
(b) In `xor` mode a hollow picture drawn on top of itself disappears. This is particularly useful when writing tools that use dynamic pictures, as one can remove a picture by merely drawing it again (see `draw_pics` in the chapter on Tool Building).
Note, however, that the penmode *only* affects the outline of objects, it has no effect on the interior patterns of filled objects.

The Pen's Pattern

The pen's pattern determines how the outline of pictures appear on the screen. Black is the default pattern for the pen.

In MacPROLOG a pattern can be defined by the user using an atom comprising 16 hexadecimal digits.

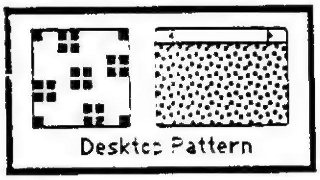
The Macintosh **Control Panel** gives an example of how patterns are determined by a grid of 8x8 pixels.



In the above diagram, we see a grey pattern defined by an 8x8 pixel grid. Its hexadecimal description would be

    'AA55AA55AA55AA55'.

(Black pixels may be represented by a binary 1, and white pixels by a binary 0. The top pixel row of the above pattern may be represented by the binary number 10101010, which is AA in hexadecimal. The rest of the pattern is represented in a similar way.)



In the above diagram, we see another pattern defined by an 8x8 grid. Its hexadecimal representation would be

    'B130031BD8C00C8D'.

24.30 `blackpen` - set the pen to black

`blackpen(`*`picture`*`)`

ARGUMENT

`picture` : a picture description

USE

To set the pen's pattern to solid black. This is the default pen pattern.

24.31 `greypen` - set the pen to grey

`greypen(`*`picture`*`)`

ARGUMENT

`picture` : a picture description

USE

To set the pen's pattern to grey.

24.32 `whitepen` - set the pen to white

`whitepen(`*`picture`*`)`

ARGUMENT

`picture` : a picture description

USE

To set the pen's pattern to white.

24.33 **penpattern** - set pen to user defined pattern

`penpattern(pattern,picture)`

ARGUMENTS

pattern	: 16-character atom representing pattern
picture	: a picture description

USE

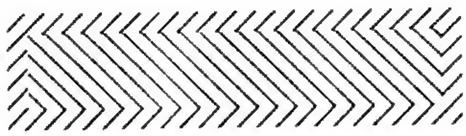
Sets the pen colour to the nominated pattern. The pattern is specified by an atom which consists of 16 hexadecimal characters.

For example, the pattern for diagonal lines is

'0102040610204080'

and the description of the object below is

```
penpattern('0102040810204080',thick(rdiag(fillbox(0,0,46,180))))
```



The box outline is drawn in the `penpattern` specified above, and the box is filled with the pattern `rdiag` which is a predefined pattern of diagonal lines going in the opposite direction (see the section on Fill Patterns below).

Fill Patterns

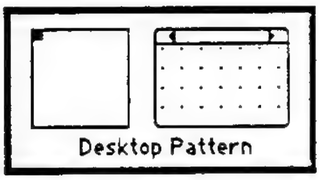
All calls to filled patterns are of the form

fill(picture) where *fill* is one of the following

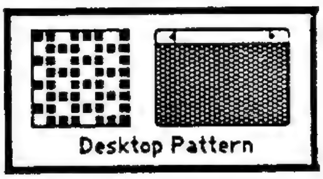
```
black
brick
boxed
check
crosses
diag
diamonds
grey
horiz
lgrey
rdiag
speckled
stripesthin
stripesthick
waves
white
alpha
beta
```

As with all transformers, the outermost fill descriptor will take priority. All fill transformers are absolute, there is no notion of 'stripier' or 'blacker'.

There are a certain number of built in patterns, but users can define their own as hexadecimal atoms in the same manner as pen patterns.



For example, the above pattern would be represented by the atom `'8000000000000000'`.



The above pattern would be represented by the atom `'DD77DD77DD77DD77'`.

24.34 **`grey` - set fill pattern to grey**

`grey(`*`picture`*`)`

ARGUMENT

`picture` : a picture description

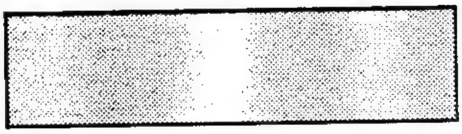
USE

Sets the fill pattern to grey.

For example

```
grey(fillbox(0,0,46,180))
```

describes



24.35 **`lgrey` - set fill pattern to light grey**

`lgrey(`*`picture`*`)`

ARGUMENT

`picture` : a picture description

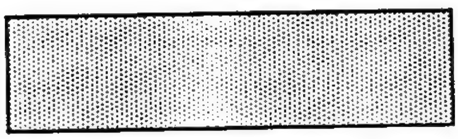
USE

Sets the fill pattern to light grey.

For example

```
lgrey(fillbox(0,0,46,180))
```

describes



24.36 **`boxed`** - set fill pattern to boxed

`boxed(`*`picture`*`)`

ARGUMENT

`picture` : a picture description

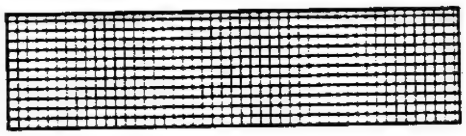
USE

Sets the fill pattern to boxed.

For example

```
boxed(fillbox(0,0,46,180))
```

describes



24.37 **`brick`** - set fill pattern to brick

`brick(`*`picture`*`)`

ARGUMENT

`picture` : a picture description

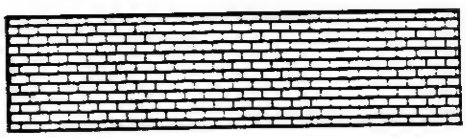
USE

Sets the fill pattern to brick.

Example

```
brick(fillbox(0,0,46,180))
```

describes



24.38 **`check`** - set fill pattern to check

`check(`*`picture`*`)`

ARGUMENT

`picture` : a picture description

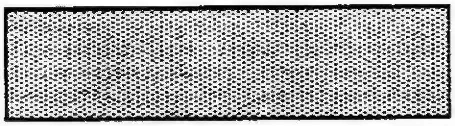
USE

Sets the fill pattern to check.

Example:

```
check(fillbox(0,0,46,180))
```

describes



24.39 **`crosses`** - set fill pattern to crosses

`crosses(`*`picture`*`)`

ARGUMENT

`picture` : a picture description

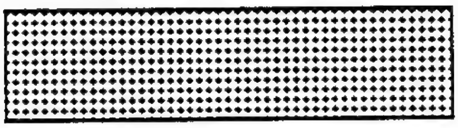
USE

Sets the fill pattern to crosses.

Example

```
crosses(fillbox(0,0,46,180))
```

describes



24.40 diag - set fill pattern to diagonal stripes

`diag(`*`picture`*`)`

ARGUMENT

picture : a picture description

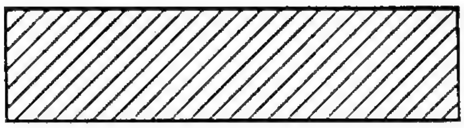
USE

Sets the fill pattern to diagonal stripes.

For example

```
diag(fillbox(0,0,46,180))
```

describes



24.41 diamonds - sets fill pattern to diamonds

`diamonds(`*`picture`*`)`

ARGUMENTS

picture : a picture description

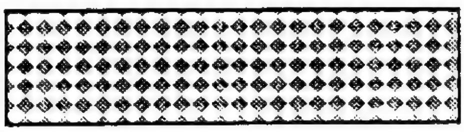
USE

Sets the fill pattern to diamonds.

Example

```
diamonds(fillbox(0,0,46,180))
```

describes



24.42 `horiz` - set fill pattern to horizontal stripes

`horiz(`*`picture`*`)`

ARGUMENT

`picture` : a picture description

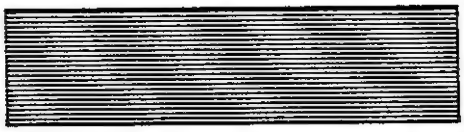
USE

Sets the fill pattern to horizontal stripes.

Example

```
horiz(fillbox(0,0,46,180))
```

describes



24.43 `rdiag` - set fill pattern to reverse diagonal stripes

`rdiag(`*`picture`*`)`

ARGUMENT

`picture` : a picture description

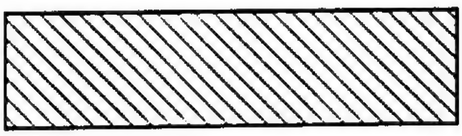
USE

Sets the fill pattern to a reverse diagonal stripes.

Example:

```
rdiag(fillbox(0,0,46,180))
```

describes



24.44 **speckled** - set fill pattern to a speckled pattern

speckled(*picture*)

ARGUMENT

picture : a picture description

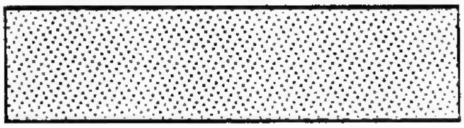
USE

Sets the fill pattern to a 'speckled' pattern.

Example

```
speckled(fillbox(0,0,46,180))
```

describes



24.45 **stripesthin** - set fill pattern to vertical thin stripes

stripesthin(*picture*)

ARGUMENT

picture : a picture description

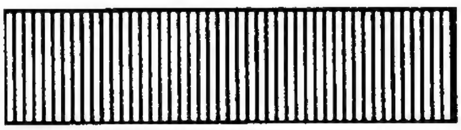
USE

Sets the fill pattern to vertical thin stripes.

Example

```
stripesthin(fillbox(0,0,46,180))
```

describes



24.46 **`stripesthick`** - set fill pattern to vertical thick stripes

`stripesthick(`*`picture`*`)`

ARGUMENT

`picture` : a picture description

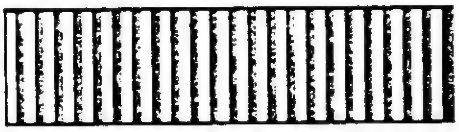
USE

Sets the fill pattern to vertical thick stripes.

Example

```
stripesthick(fillbox(0,0,46,180))
```

describes



24.47 **`waves`** - sets fill pattern to waves

`waves(`*`picture`*`)`

ARGUMENT

`picture` : a picture description

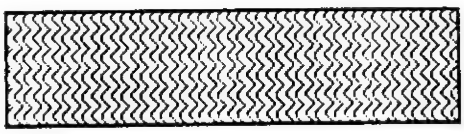
USE

Sets the fill pattern to waves.

Example

```
waves(fillbox(0,0,46,180))
```

describes



24.48 **`white`** - set fill pattern to white

`white(`*`picture`*`)`

ARGUMENT

`picture` : a picture description

USE

Sets the fill pattern to white. Placing a white filled picture over an existing picture will obscure the picture underneath.

Example

```
white(fillbox(0,0,46,180))
```

describes

24.49 **`alpha`** - set fill pattern to alpha

`alpha(`*`picture`*`)`

ARGUMENT

`picture` : a picture description

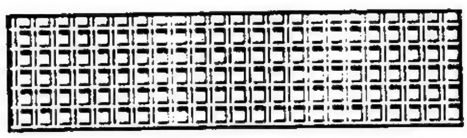
USE

Sets the fill pattern to alpha.

Example

```
alpha(fillbox(0,0,46,180))
```

describes



24.50 **`beta`** - set fill pattern to beta

`beta(`*`picture`*`)`

ARGUMENT

`picture` : a picture description

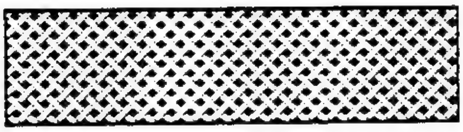
USE

Sets the fill pattern to beta.

Example

```
beta(fillbox(0,0,46,180))
```

describes



24.51 **`fillpattern`** - set fill pattern to user defined pattern

`fillpattern(`*`pattern,picture`*`)`

ARGUMENTS

`pattern` : atom representing pattern
`picture` : a picture description

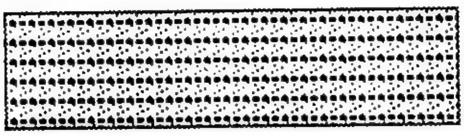
USE

Sets the pen pattern to the nominated pattern. The pattern is specified by an atom which consists of 16 hexadecimal characters, as described earlier.

For example

```
greypen(fillpattern('024008208004',fillbox(0,0,46,180)))
```

describes:



24.52 **`invert` - invert a picture**

`invert(`*`picture`*`)`

ARGUMENTS

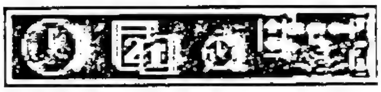
`picture`	: a picture description

USE

Inverts any filled picture within the scope of the `invert` operator. All the black pixels become white and all the white pixels become black.

The following inverted resource picture is described by

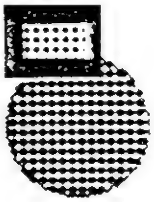
```
invert(picture(258,307,285,455,resource(-15968)))
```



and

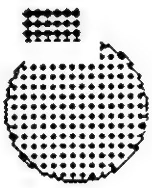
```
crosses([invert(greypen(double(filloval(20,0,60,60)))),
                         thick(fillbox(0,0,30,40))])
```

describes



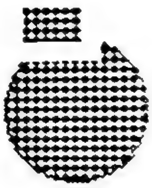
where the filled oval has been inverted, but not the filled box. However, in

```
crosses([greypen(double(filloval(20,0,60,60))),
                         invert(thick(fillbox(0,0,30,40)))])
```



it is the filled box that has been inverted (the black pen becomes white, not transparent). Finally, we invert both the filled box and the filled oval.

```
invert([crosses([greypen(double(filloval(20,0,60,60))),
                         thick(fillbox(0,0,30,40))])])
```



24.53 User Defined Graphical Forms

One very powerful feature of the MacPROLOG graphics package is the ability of programmers to define their own graphical forms.
These can be new picture descriptors, new transformation descriptors, text displaying descriptors etc. A user defined form is then treated on an equal basis with existing system forms, enabling users to extend the Graphic Description Language.

These user defined graphics descriptors, like system descriptors, are names of programs that will be called by the MacPROLOG graphics evaluator when it is drawing a picture described by some picture term. These programs have a special format, and cannot be called directly.

In order to define a new graphic k-argument descriptor `gf`, simply define `gf` as a k+1 argument relation, where the last argument is a variable which `gf` will instantiate to a GDL description. When MacPROLOG is drawing a picture and it encounters the descriptor `gf` with arguments $Arg_1, \ldots, Arg_k$ it will simply execute the call

```
gf(Arg1,...,Argk,Description)
```

and then continue the drawing operation using the *`Description`* which `gf` has returned.

Note that *`Description`* can contain other user defined forms, including other uses of `gf`. This allows recursively defined user forms. But be very careful when defining recursive forms to ensure termination.

For example, a star, defined as a sequence of crossing lines, whose position is specified relative to a point could be implemented by the program

```
trans_star(Y,X,trans(Y,X,lines([(150,150),(225,180),
                  (175,105),(175,195),(225,120)]))).
```

In this case

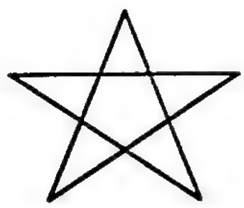
```
lines([(150,150),(225,180),(175,105),(175,195),(225,120)])
```

describes the star, on which `trans` has the effect of translating through `Y`, `X`.

The GDL term

```
trans_star(20,50)
```

describes this star translated 20 pixels down and 50 pixels across.

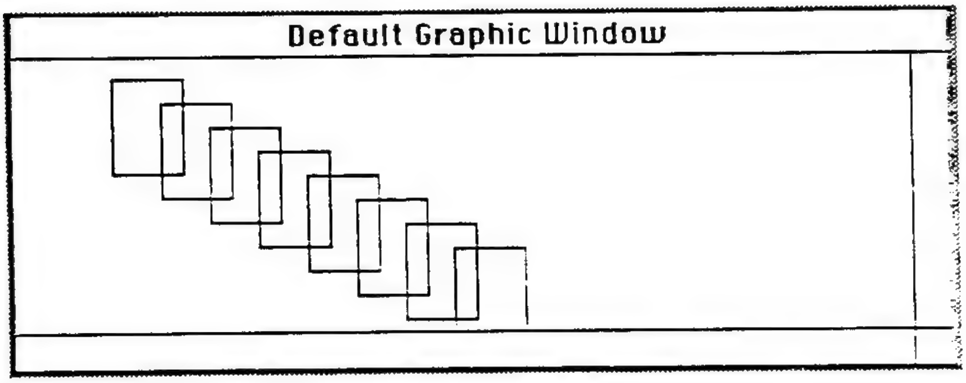


An example of a recursive user-form is

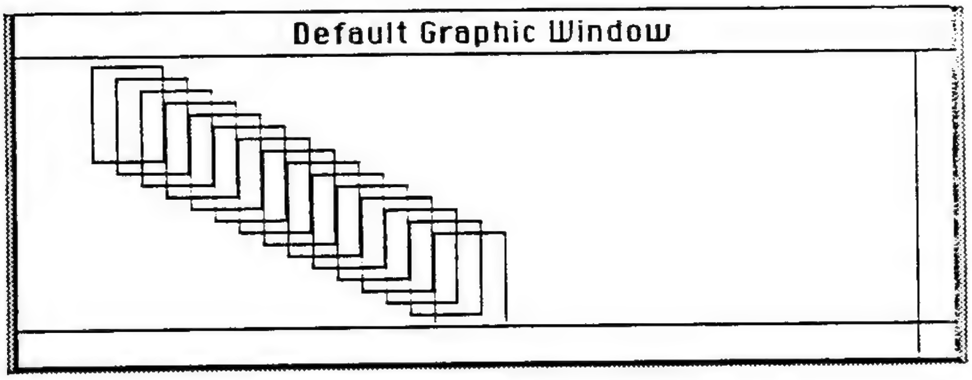
```
movable_box(DY,DX,0,box(DY,DX,40,30)).
movable_box(DY,DX,N,[box(DY,DX,40,30),trans(DY,DX,Desc)]):-
        0 < N,
        N1 is N--1,
        movable_box(DY,DX,N1,Desc).
```

Below are some examples of using this user form with different initial arguments.

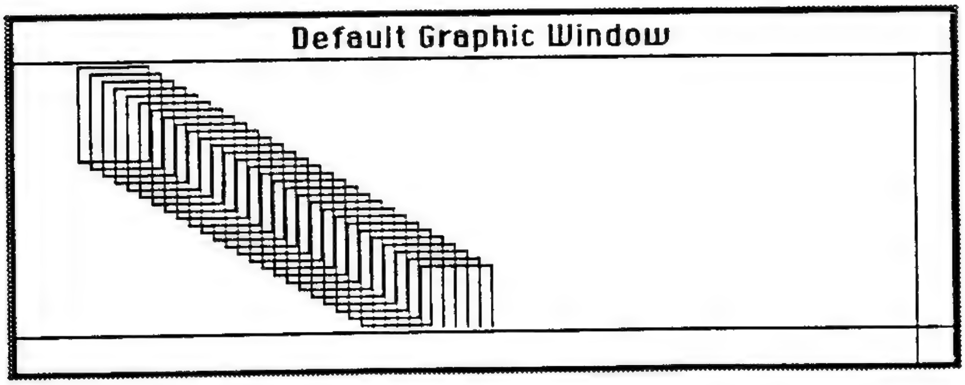
```
add_pic(movable_box(10,20,7))
```




```
add_pic(movable_box(5,10,14))
```




```
add_pic(movable_box(3,5,28))
```

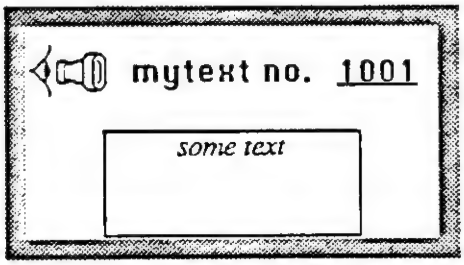


The combination of user defined forms and text provides a powerful basis for presenting structured information and pictures. Consider the following rather complex definition

```
mytext(Num,Box1,Text,Desc):-
        inset_box(Box1,(0,-20),Temp),
        offset_box(Temp,(20,0),box(T1,L1,D1,W1)),
        inset_box(box(T1,L1,D1,W1),(-1,-1),Border),
        inset_box(Box1,(-30,-60),Expanded),
        offset_box(Expanded,(10,10),box(T2,L2,D2,W2)),
        ++(20,T2,TextT),
        ++(40,L2,TextL),
        text_width('mytext no.','Chicago',12,0,Width),
        ++(Width,TextL,NextL),
        pname(Num,Con),
        Desc=[thick(greypen(Expanded)),Border,textbox('Times',12,2
,T1,L1,D1,W1,1,Text),icon(T2,L2,32,32,Num),text('Chicago',12,0
,TextT,TextL,'mytext no.'),text('Chicago',12,4,TextT,NextL,Con)].
```

The picture described by the term

`mytext(1001,box(10,10,40,60),'some text')` is shown below



`inset_box` and `offset_box` are primitives that manipulate `box` terms, and `text_width` calculates the width of a given atom. (See the chapter on Tool Building).

A more efficient way of implementing the above user defined form would be to do less manipulation of arguments within the program that defines the form. This is because as the above definition stands, MacPROLOG would have recalculate all the relative positions of the sub-pictures in the user defined form every time one of these pictures was drawn. Given that most pictures are fairly static once defined, this is unnecessary. By calculating the relative positions once, on creation of the new picture, we can eliminate these repeated recalculations. We could either pass in all these precalculated arguments explicitly, or save them as properties or clauses and access them from within the user defined form , e.g.

```
mytext2(Index,Desc):-
        picture_info(Index,Expanded,Border,Text,Num,Con,
              T1,L1,D1,W1,T2,L2,TextT,TextL,NextL),
        Desc=[thick(greypen(Expanded)),Border,
              textbox('Times',12,2,T1,L1,D1,W1,1,Text),
              icon(T2,L2,32,32,Num),
              text('Chicago',12,0,TextT,TextL,'mytext no.'),
              text('Chicago',12,4,TextT,NextL,Con)].
```

By implementing a specialized tool that used the portable editor as described in the chapter on Tool Building, you could allow users to reedit the text of these user defined forms .

25 Picture Manipulation

Every graphics window has an associated list of pictures. Each picture on the list has:

1) A **name**, unique to that window, to identify the picture.

2) A picture **description** in the GDL.

3) A local **origin** represented as a point pair (Y, X). This point pair represents the shift that needs to be applied to the picture description in order to draw it as currently displayed in the window.

4) a **selection** flag (1 if the picture is currently selected, 0 if not).

The primitives in this chapter enable you to add to and manipulate the associated list of pictures for a graphic window.

25.1 add_pic - add a new picture to a graphic window and draw it

```
add_pic(desc)
add_pic(name, desc)
add_pic(window, name, desc)
add_pic(window, name, desc, org)
```

ARGUMENTS

desc	: a picture description in GDL
name	: atom, name of picture element
window	: atom, name of graphic window
org	: a point pair of the form (*Y*, *X*) , amount of initial translation

USES

The picture described by `desc` is immediately drawn in the drawing area of the window. If the window's viewing pane is over that part of the drawing area, the picture immediately becomes visible, and it will be drawn on top of any pictures which it overlaps. Its description is recorded in the picture list for the window. Initially it will not be 'selected'.

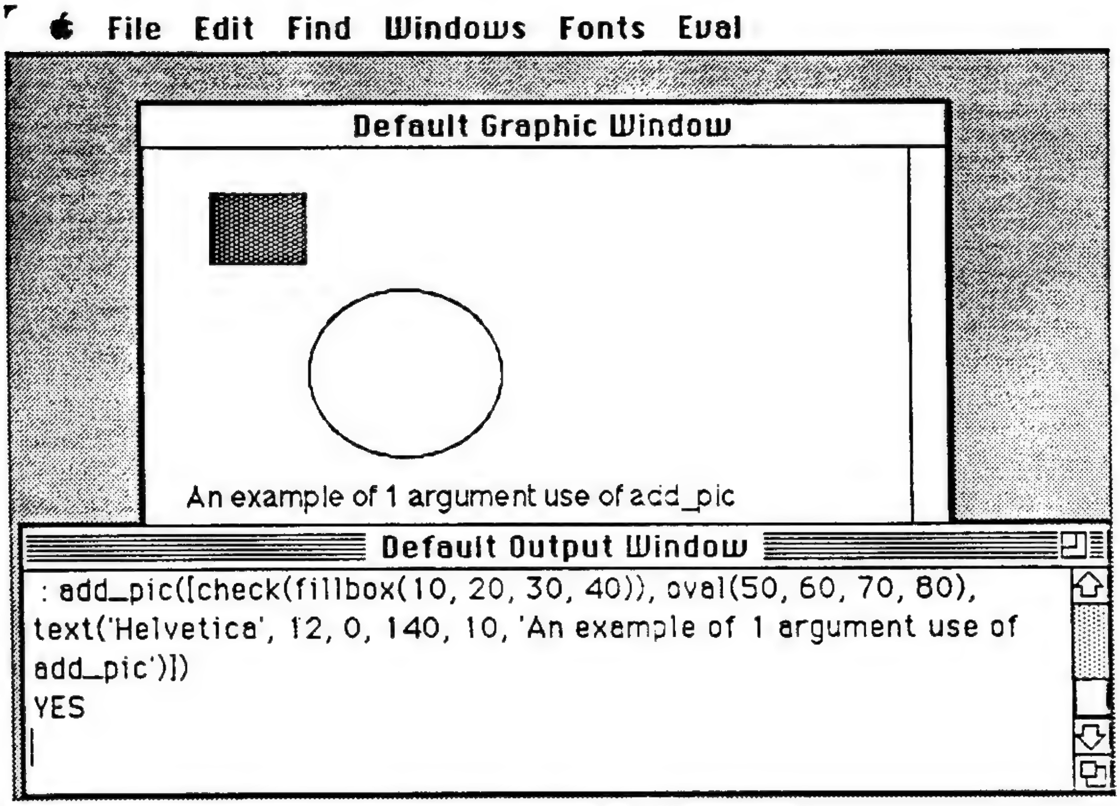
NOTE At any given time, only a small part of a graphic window's drawing area is visible in the viewing pane. By scrolling the window, the viewing pane can be moved over the drawing area. If you add a picture outside the currently visible part of the drawing area, then you will have to scroll the viewing pane of the window to see it.

1. One argument use: Adds the picture entity described by `desc` to the **Default Graphic Window**, and generates a unique name for the picture using a `gensym(picture,`*Name*`)` call.

2. Two argument use: Adds the named picture entity described by `desc` to the **Default Graphic Window**.

3. Three or Four argument use: Adds the named picture entity described by `desc` to the named graphic `window`. If `org` is given, then the picture will also be translated by the amount given, otherwise its local origin will be set to (0,0).

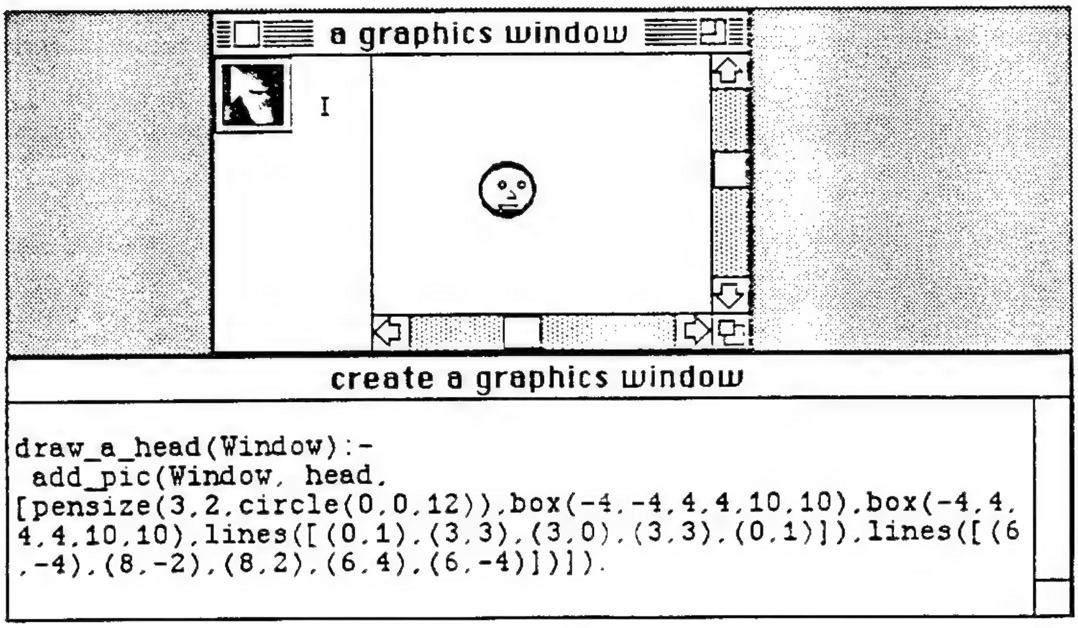
Example



As another example, the following picture is drawn by the call

```
draw_a_head('a graphics window')
```

where `'a graphics window'` has previously been created by a call to `wgcreate` (see the chapter on Graphic Windows).



25.2 **add_qpic - add a new picture in Quickdraw format**

```
add_qpic(desc)
add_qpic(name, desc)
add_qpic(window, name, desc)
add_qpic(window, name, desc, org)
```

ARGUMENTS

desc	: a picture description in GDL
name	: atom, name of picture element
window	: atom, name of graphic window
org	: a point pair of the form `(Y, X)`, amount of initial translation

USES

The use of `add_qpic` is identical to that of `add_pic` except that the picture described by *desc* is converted to Quickdraw format first. The advantage of this is that subsequent redrawing of the picture (during screen refreshes and scrolling, for example), is much quicker. However, the disadvantage is that any 'structure' the picture may have had is lost - the Quickdraw picture may only be manipulated as a single unit.

25.3 **del_pic - remove a picture**

```
del_pic(window, name)
```

ARGUMENTS

window	: atom, name of graphic window
name	: atom, name of picture

USE

The named picture is removed from the graphic window's picture list. If the picture is currently visible in the viewing pane, then it will disappear when the Mac next refreshes its display (see section on Advanced Picture Drawing).

If there is no such picture in the window, then `del_pic` does nothing and the call succeeds.

25.4 `del_sels` - remove selected pictures

`del_sels(window)`

ARGUMENT

window : atom, name of graphic window

USE

The pictures which are currently selected in the named window are removed from the graphic window's picture list. If the pictures are currently visible in the viewing pane, they will disappear when the Mac next refreshes its display (see section on Advanced Picture Drawing).

25.5 `del_all` - remove all pictures

`del_all(window)`

ARGUMENT

window : atom, name of graphic window

USE

The graphic window's picture list is emptied, and the viewing pane is immediately cleared.

25.6 `del_pic_num` - remove Nth picture

`del_pic_num(window,position)`

ARGUMENTS

window : atom, name of graphic window
position : integer, position of picture on picture list

USE

To remove the *N*th picture from the picture list of a graphic window. If *position* is 1, the last picture to be added to the list is deleted. The picture is visibly removed when the Mac next refreshes its display.

If *position* is greater than the number of pictures in the named window, then `del_pic_num` does nothing and the call succeeds.

The primitive `get_all_pics` will return the ordered list of all the pictures in the picture list for a graphic window.

To remove the most recently added picture from a window (for example in generate and test graphic traces), use the call

```
del_pic_num(window,1).
```

25.7 get_pic - retrieve picture information

```
get_pic(window,name,desc)
get_pic(window,name,desc,org)
get_pic(window,name,desc,org,select)
```

ARGUMENTS

window	: atom, name of graphic window
name	: atom, name of picture element
desc	: variable, will be bound to the picture's description
org	: variable, will be bound to a point pair of the form (*Y*, *X*)
select	: variable, will be bound to the picture's selection flag (0 or 1)

USE

The named picture's description, origin, and selection flag are retrieved.

EXAMPLE

If the picture named `head` had been added to the window `graphic` using the example `add_pic` call given above, then the call

```
get_pic(graphic,head,Desc)
```

will succeed, with *Desc* bound to

```
[pensize(3,2,circle(0,0,12)),box(-4,-4,0,0,10,10),
      box(-4,4,0,8,10,10),lines([(1,0),(3,3),(0,3),(3,3),(1,0)]),
      lines([(-4,6),(-2,8),(2,8),(4,6),(-4,6)])]
```

25.8 chg_pic - alter a picture's description and redraw it

```
chg_pic(window,name,newdescription)
```

ARGUMENTS

window	: atom, name of graphic window
name	: atom, name of picture element
newdescription	: picture description

USE

The named picture has its current description replaced by *newdescription*.. The old picture will be erased and the new picture displayed when the Mac next refreshes its display.

25.9 `shift_pic` - translate a picture

`shift_pic(window, name, amount)`

ARGUMENTS

window	: atom, name of graphic window
name	: atom, name of picture element
amount	: a point pair of the form (*Y*, *X*)

USE

The named picture entity's description in GDL is translated by the amount indicated, where *amount* is a term of the form

(*down*, *across*)

where *down* and *across* are integers. The integers may be positive, negative or zero. This translation is achieved by manipulating the picture's local origin; the picture's description remains unaltered. The old picture is removed and the translated picture displayed when the Mac next refreshes its display.

25.10 `shift_pics` - translate a list of pictures

`shift_pics(window, names, amount)`

ARGUMENTS

window	: atom, name of graphic window
names	: list of names of pictures
amount	: a point pair of the form (*Y*, *X*)

USE

All the pictures whose names appear on the *names* list are translated by the amount indicated, where *amount* is a term of the form

(*down*, *across*)

with *down* and *across* being integers. The integers may be positive, negative or zero. This translation is achieved by manipulating each picture's local origin; the pictures' descriptions remain unaltered. The old pictures are removed and the translated pictures displayed when the Mac next refreshes its display.

25.11 `pic_frame` - get a picture's enclosing rectangle

`pic_frame(desc, frame)`

ARGUMENTS

desc	: picture description
frame	: variable, will be bound to a term of the form `box(T,L,D,W)`

USE

To find out the minimum rectangle that encloses a picture description.

25.12 `pic_centre` - get a picture's local centre

`pic_centre(desc, centre)`

ARGUMENTS

desc	: picture description
centre	: variable, will be bound to a point pair of the form `(Y,X)`

USE

To find out the local centre for a named picture. The picture's local origin will *not* be taken into account, so if the picture has been translated by a call to `shift_pic`, you will have to adjust the *centre* accordingly.

This call is useful prior to scaling a picture about its own centre.

25.13 `sel_pics` - select a list of pictures and highlight them

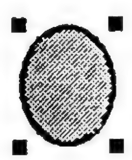
`sel_pics(window, names)`

ARGUMENTS

window	: atom, name of graphic window
names	: list of atoms, names of pictures

USE

The named pictures are selected, and immediately highlighted on the screen. This highlighting is done by surrounding the picture with four black marks, as shown below.



25.14 `desel_pics` - deselect a list of pictures

`desel_pics(`*`window, names`*`)`

ARGUMENTS

window	: atom, name of graphic window
names	: list of atoms, names of pictures

USE

The named pictures are deselected, and any highlighting marks are removed.

25.15 `sel_all` - select all the pictures in a window and highlight them

`sel_all(`*`window`*`)`

ARGUMENT

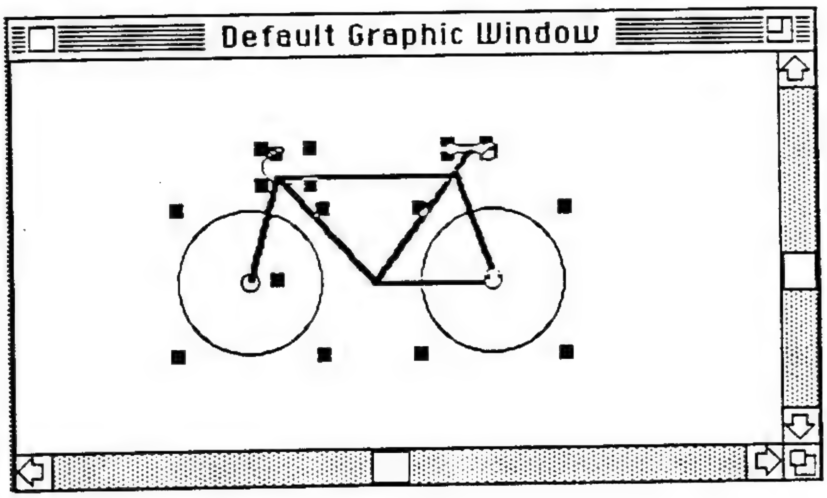
window	: atom, name of graphic window

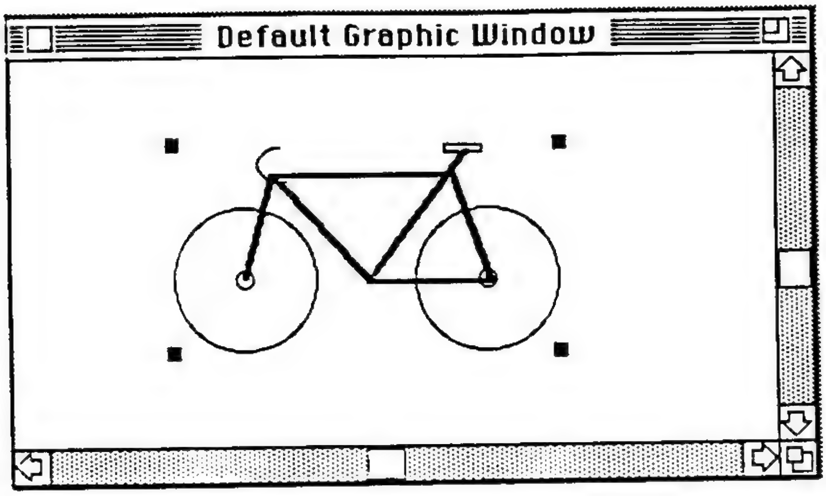
USE

To select all the pictures in the picture list of a graphic window, and display them highlighted.

EXAMPLE

If we apply `sel_all` to the examples of the bike drawn in the **Default Graphic Window** (see the initial chapter on the GDL), we obtain two different displays, one for the bike comprised of 6 parts and the other for the bike comprised of one part.





25.16 `desel_all` - deselect all the pictures in a window

`desel_all(window)`

ARGUMENT

window	: atom, name of graphic window

USE

To deselect all the pictures in the picture list of the graphic window and display them unhighlighted.

25.17 `get_sel_pics` - return the list of names of selected pictures

`get_sel_pics(window,names)`

ARGUMENTS

window	: atom, name of graphic window
names	: variable, will be bound to list of picture names

USE

To find out the names of all currently selected pictures within a graphic window.

25.18 **`get_desel_pics`** - return the list of names of deselected pictures

`get_desel_pics(`*`window, names`*`)`

ARGUMENTS

`window`	: atom, name of graphic window
`names`	: variable, will be bound to list of picture names

USE

To find out the names of all currently non-selected pictures within a graphic window.

25.19 **`get_all_pics`** - get list of pictures in a window

`get_all_pics(`*`window, names`*`)`

ARGUMENTS

`window`	: atom, name of graphic window
`names`	: variable, will become list of names of pictures

USE

To get the names of all the pictures in the picture list of a graphic window. The *`names`* argument will be bound to a list of atoms that are the names of the pictures in the named *`window`*.

The order of the names in the list corresponds to the order of the pictures in the window, with the first picture in the list being the topmost picture in the window, and the last picture in the list being the bottommost picture in the window.

Picture ordering

The following three primitives reorder the picture list and then redraw the display according to the new picture list. Whilst this does not affect any individual picture description, it can result in a different display as previously hidden pictures may now be visible, and previously visible pictures may now be hidden or partially hidden.

25.20 `reverse_pics` - reverse the picture list of a window

`reverse_pics(`*`window`*`)`

ARGUMENT

window : atom, name of graphic window

USE

To reverse the order of the picture list for a named graphic window. Where pictures overlap, the bottommost pictures become the topmost, and the topmost the bottommost. The window is redrawn when the Mac next refreshes its display.

25.21 `bring_to_front` - bring pictures to the front of the picture list

`bring_to_front(`*`window`*`)`
`bring_to_front(`*`window, names`*`)`

ARGUMENTS

window : atom, name of graphic window
names : list of atoms, names of pictures

USES

1. One argument use: To bring to the front of the picture list all the currently selected pictures in a graphic window. These are immediately redrawn on top of any unselected pictures they may overlap.

2. Two argument use: To bring to the front of the picture list all the pictures in *names* . These are immediately drawn on top of any pictures they may overlap.
Note that the pictures will be drawn corresponding to the order of their names on the *names* list, so that the first picture named in *names* will become the new topmost picture.

25.22 **`send_to_back`** - send pictures to the back of the picture list

```
send_to_back(window)
send_to_back(window,names)
```

ARGUMENTS

window	: atom, name of graphic window
names	: list of atoms, names of pictures

USES

1. <u>One argument use:</u> To send to the back of the picture list all the currently selected pictures in the graphic window. This has the same effect as sending all the currently deselected pictures to the front of the picture list. The deselected pictures are immediately redrawn on top of any selected pictures they may overlap.

2. <u>Two argument use:</u> To send to the back of the picture list all the pictures in *names*. This has the same effect as sending all the pictures not in *names* to the front of the picture list. These pictures not in *names* are then immediately redrawn, possibly obscuring some of the named pictures.

Manipulating pictures using Edit commands

The **Edit** menu of the programming environment can be used to manipulate pictures when a graphic window is the front window on the screen, as follows:

Cut	removes all the selected pictures from the window, and places them on the clipboard.
Copy	copies all the selected pictures in the window onto the clipboard.
Paste	copies any pictures on the clipboard into the window.
Clear	removes all the selected pictures from the window, without affecting the clipboard's contents.
Select all	selects all the pictures in the window.

The `cut`, `copy`, `clear` and `paste` primitives described in the chapter on Window Handling also manipulate pictures.

The **Clipboard format** on the **Defaults...** dialogue determines the format used to represent pictures on the clipboard.

There are two formats:

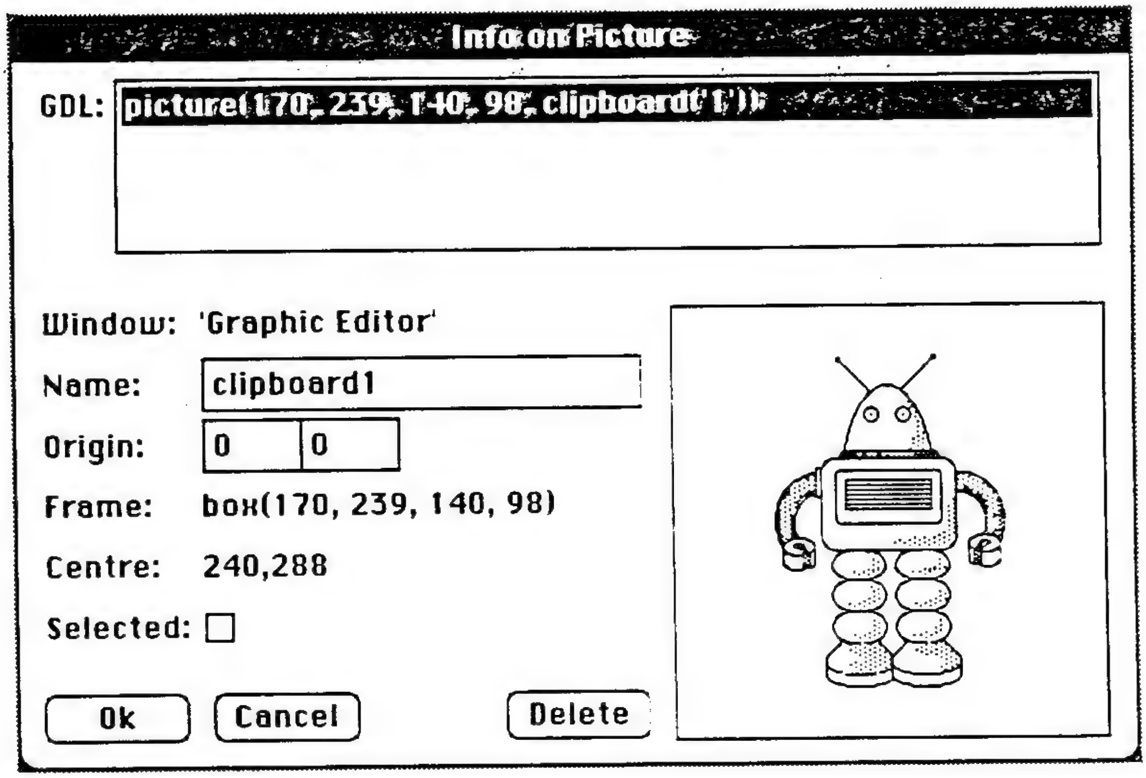
1. **MacPROLOG™** format is used for internal cutting and pasting. In this format, it is the text of the GDL descriptions that are used rather than the pictures they describe, so preserving any structural information. If you **Cut** or **Copy** to the clipboard using this format and you then do **Show clipboard**, you will see the text of the GDL description of the picture on the clipboard.

2. **QuickDraw** is used for importing or exporting pictures to and from other applications. In this format, the GDL descriptions are converted into a single QuickDraw picture whose internal structure is unknown. This QuickDraw picture is displayed in the clipboard.

Pasting of an imported picture (through the scrapbook or clipboard) is always in QuickDraw format, but pasting of an internal MacPROLOG picture, say from one graphic window to another, will duplicate the MacPROLOG description, name and origin of the picture or pictures.

EXAMPLE

The picture shown in the 'Info on Picture' dialogue below was copied to the clipboard from the Scrapbook, and then pasted into a graphic window named **Graphic Editor**.



Its description is the term

```
picture(170,239,140,98,clipboard('1'))
```

where `(170,239,140,98)` represents its size and position in the window, and `clipboard('1')` identifies it as the second picture to be pasted into a graphic window in this session.

MacPROLOG keeps track of all these pictures until the session is terminated, and assigns picture names to them as needed (`clipboard1` in the above example).

You can save pictures as resource items using the `save_pic` primitive explained in the chapter on Advanced Drawing.

Extensions to picture descriptions

Before adding a GDL picture description to a window, MacPROLOG calculates the size of the minimum rectangle that encloses the picture. The corners of this rectangle are marked whenever the picture is currently selected, and this rectangle is called the picture's frame. This frame is used frequently by the graphics system, but it takes time to compute.

In addition to a GDL picture description, as described in the chapter on GDL, MacPROLOG allows two other 'extended forms' of picture description which do not require the enclosing rectangle to be computed by the system.

(1) Suppose `Desc` is a GDL picture description, and `box(T,L,D,W)` describes the minimum enclosing rectangle for the picture. The term

```
framed_picture(box(T,L,D,W),Desc)
```

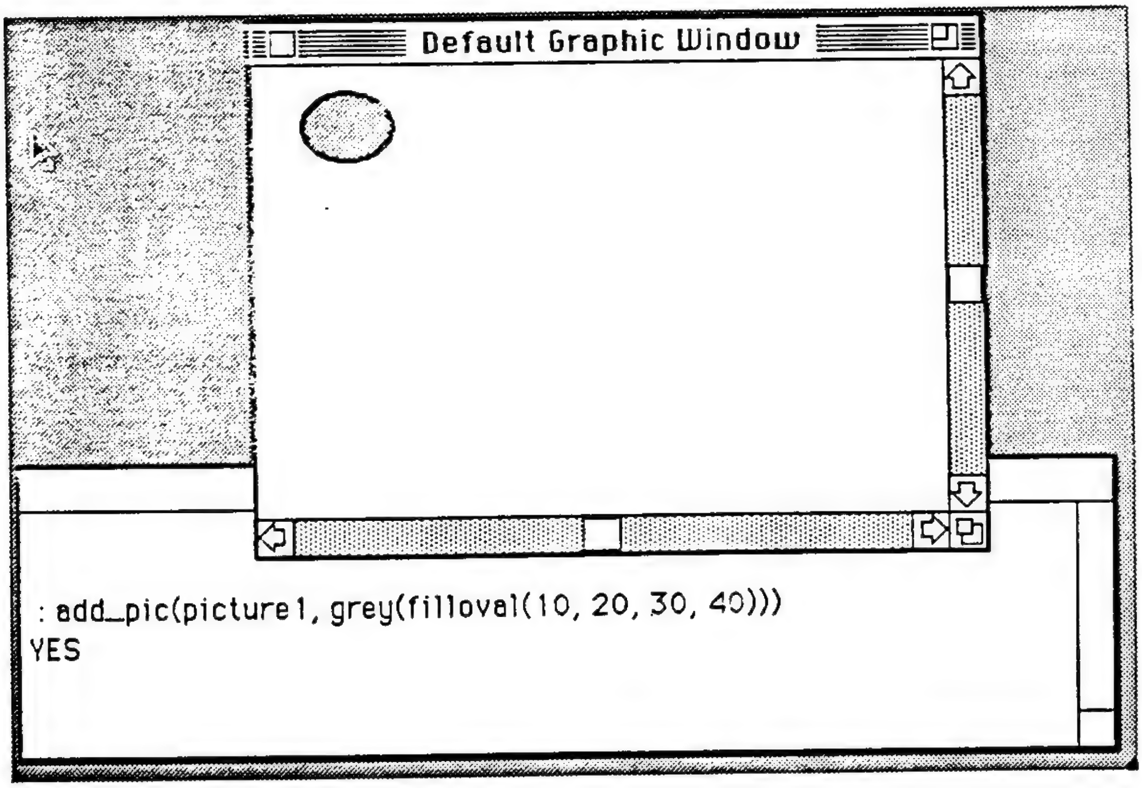
can be used as an alternative description of the picture. MacPROLOG will take `box(T,L,D,W)` as the enclosing rectangle and will not bother to compute the rectangle.

Example:

```
add_pic(picture1,framed_picture(box(10,20,30,40),
                    grey(filloval(10,20,30,40))))
```

is a slightly faster way of adding a grey filled oval to the **Default Graphic Window** than

```
add_pic(picture1,grey(filloval(10,20,30,40))).
```



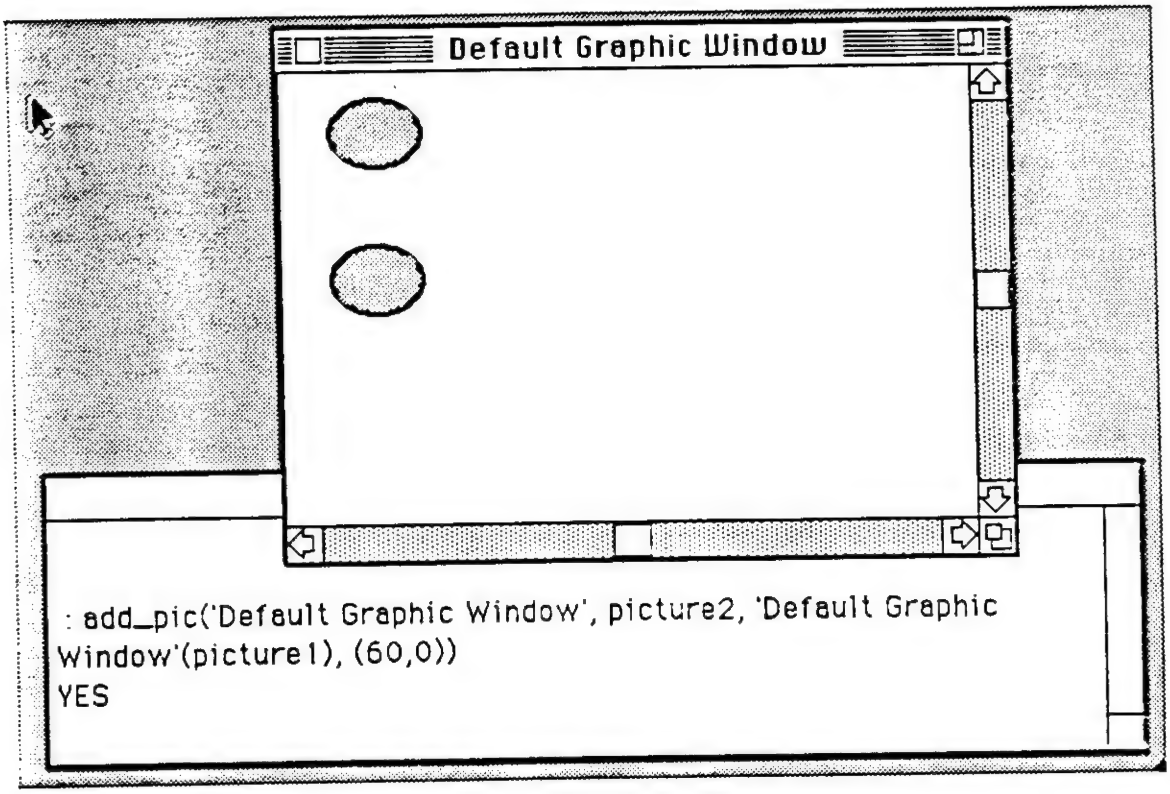
(2) Suppose that *name* is the name of a picture which is in the associated list of pictures of the graphic window called *window*. The descriptor term

```
window(name)
```

can be used to draw the same picture in another window. Because MacPROLOG has already calculated the enclosing rectangle for the picture, it will not be recalculated.
For example the call

```
add_pic('Default Graphic Window',picture2,
        'Default Graphic Window'(picture1),(60,0))
```

will result in a quick copy of the `picture1` picture of the **Default Graphic Window** being added as `picture2` to the same window at the origin (60,0).



The call

```
add_pic(mywindow,picture3,
        'Default Graphic Window'(picture2))
```

will result in a copy of the picture `picture2` from the **Default Graphic Window** being added to `mywindow` as `picture3`. This call assumes that there is no picture called `picture3` already in `mywindow` (because picture names must be unique within windows).

These extensions are also valid for the `add_pic` and `chg_pic` primitives and in the format descriptors `pbutton` and `pcheck` for dialogue items.

26 Graphic Windows

A graphic window differs from a text window in that it is used to display **graphical objects**, such as lines, circles, and boxes rather than plain text, although it can also have edit and text fields, rather like dialogues.

True to its multi-window environment, there can be any number of graphic windows in MacPROLOG™ at any time. They can be moved, dragged or tidied just like text windows, and the general window handling primitives `wtype`, `wrename`, `whide`, `windows`, `cleanup`, `wfront`, `wshow`, `wvis` and all the editing primitives also extend to graphic windows.

A **graphic window** can have all the usual controls associated with it: a *goaway box* for hiding the window, a *grow box* for resizing it, and a *zoom box* for making the window fill the screen. The **Default Graphic Window** is a graphic window provided by the MacPROLOG system. Additional graphic windows can be created using the primitive `wgcreate`.

Graphic windows are usually split into 2 panes by an adjustable vertical **split** line. On the left of this line is the **tool pane**, and on the right is the **viewing pane**.

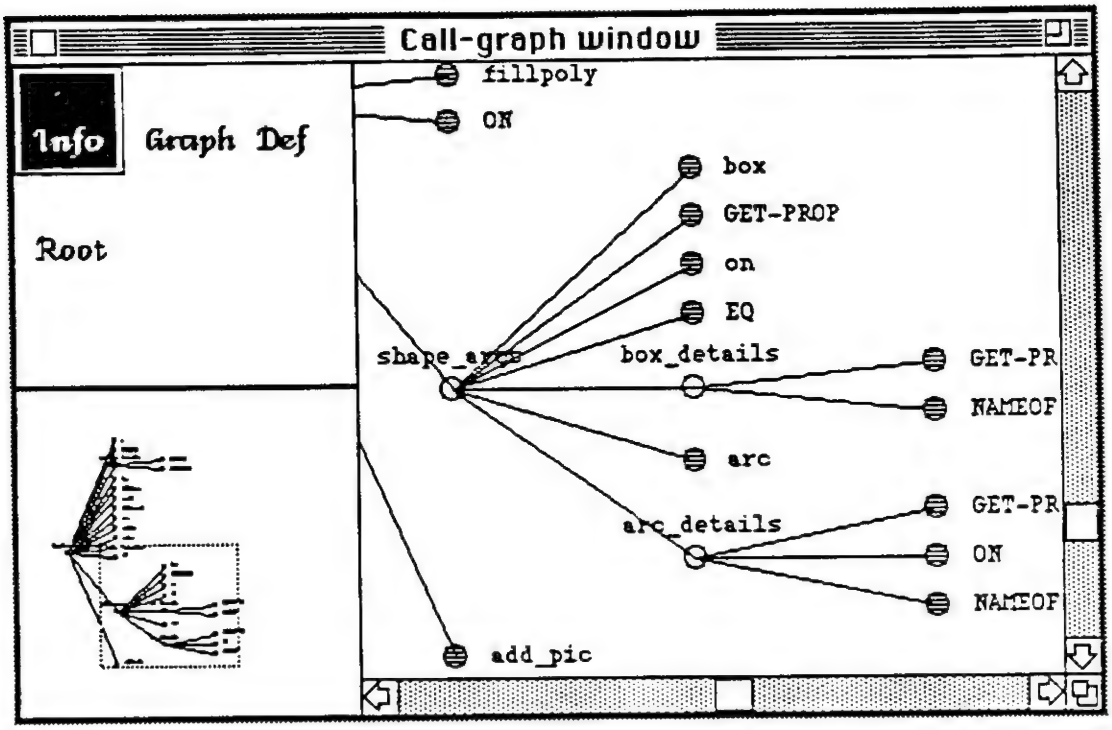
The **viewing pane** is a viewport onto the potentially very large **drawing area** of the window. The window's vertical and horizontal scroll bars allow the viewing pane to be moved over the drawing area.

The **tool pane** is fixed, it cannot be scrolled. It contains the graphical images of the tools associated with the window, and an optional **viewer**. The tool images are arranged in the tool pane in vertical columns. The number of columns used may be set by the user.

The **viewer** displays a scaled down overview of the entire drawing area. All the pictures in the drawing area (apart from text pictures) are shown in this overview. The current position of the viewing pane in the drawing area is represented by a grey outlined rectangle. This rectangle may be dragged to a new position, and the viewing pane will be moved to the corresponding position over the drawing area. It is thus a fast alternative to using scroll bars.

Tools are MacPROLOG programs that can be invoked to draw, get information or otherwise manipulate the pictures in the window. Tools have a role similar to the pull-down menu commands. A tool is selected by clicking on its graphical representation in the tool pane, and its program is then invoked by clicking in the viewing pane. The drawing area coordinates of the mouse click are passed to the tool's program. MacPROLOG provides a library of tool programs for use in your own graphic windows: these are described later in this chapter. You may also define your own tools: see the chapter on Tool Building.

MacPROLOG contains two graphic windows in its programming environment. The first, the **Default Graphic Window** has been mentioned in the chapter on Picture Manipulation. The second, the **Call-graph window**, is shown below. It is used to display and interact with call-graphs of program relations, which assists in program development. Its use is described in the LPA MacPROLOG™ Environment Guide.



The tools for this window are

Info	get information on a relation node
Graph	generate a call graph for relation node
Def	find definition of relation node
Root	find root (first occurrence) of user node

A double click on any of the tools will generate a dialogue about the call graph.

In the viewer at the bottom of the tool pane is a scaled down picture of the entire call graph.

As with all graphic windows, by clicking and dragging on the split line you can adjust the width of the tool pane. This can also be achieved by selecting '**Window details...**' from the **Windows** menu, and editing the split field numerically. This dialogue also allows you to change the number of tools per row in the tool pane (i.e. the number of tool columns), switch the viewer on or off, and expand or shrink the maximum size of the drawing area associated with the **Call-graph window**.

All these attributes are explained later in this chapter.

Window primitives

26.1 `wgcreate` - create a graphic window

`wgcreate(window,t,l,d,w,split,maxx,maxy,vis,go)`

ARGUMENTS

window	: atom, name of new window
t	: integer, position of top of window as number of pixels down from top of Mac display
l	: integer, position of left of window as number of pixels in from left of Mac display
d	: integer, depth of window in pixels
w	: integer, width of window in pixels
split	: integer, width of tool area
maxx	: integer, maximum horizontal coordinate of drawing area
maxy	: integer, maximum vertical coordinate of drawing area
vis	: integer, 0 or 1
go	: integer, 0 or 1

USES

Creates a new graphic window (a window of type `grap`).

The initial position and size of the window on the screen is indicated by the *t, l, d, w* parameters.

The *split* determines the initial width of the tool pane, and is adjustable (using the `gsplit` primitive).

The maximum horizontal and vertical coordinates, *maxx* and *maxy*, determine the initial size of the drawing area. These can be altered (using the `gmax` primitive, or the **Window details** dialogue from the **Windows** menu). Suggested values are between 300 and 500.

The total size of the drawing area of the window is the coordinate range

$$-maxy \le Y \le maxy \qquad -maxx \le X \le maxx.$$

The initial position of the viewing pane over the drawing area is the rectangle

`(0,0,d,(w - split))`

i.e. the viewing pane initially has the point (0,0) at its top left hand corner.

If the *vis* argument is 0 then the new window is initially hidden; if it is non-zero, then it is initially the front window, and will be made visible at the first system refresh.

If the *go* flag is 1 then a goaway box is associated with the window, and if it is 0 there is no goaway box.

On creation, the **viewer** associated with the window will not be visible, and no tools are associated with the window.

PRAGMATICS

The `t l d w` parameters determine the size and position of the whole graphic window on the screen, just as with any other type of window. You can of course resize or reposition the window using the standard Mac mouse operations.

It is the *maxx* and *maxy* parameters which determine the size of the window's drawing area. This means that you will only be able to see pictures drawn within these bounds. If you set *maxx* and *maxy* to be 100, say, and then draw something at (200,300), don't be surprised if it doesn't appear in your window! (However, if you were to subsequently extend the drawing area, using `gmax` or **Window details**, you could then bring the picture into view.)

You can of course choose to use the whole coordinate plane (i.e. give a value of 32767 for *maxx* and *maxy*) as the drawing area, but it is unlikely that you will need all that space. The drawing area is how much you can 'see' by scrolling. If it is very large, you may have trouble 'finding' graphical objects that you have drawn! The viewer is also affected - because it displays a scaled down version of the whole drawing area, you will get more or less detail depending on the size of the drawing area. To start with we recommend specifying about 300 in each direction, and by experimentation you will find what is best for your own use.

26.2 `gviewer` - switch the viewer

`gviewer(window, switch)`

ARGUMENTS

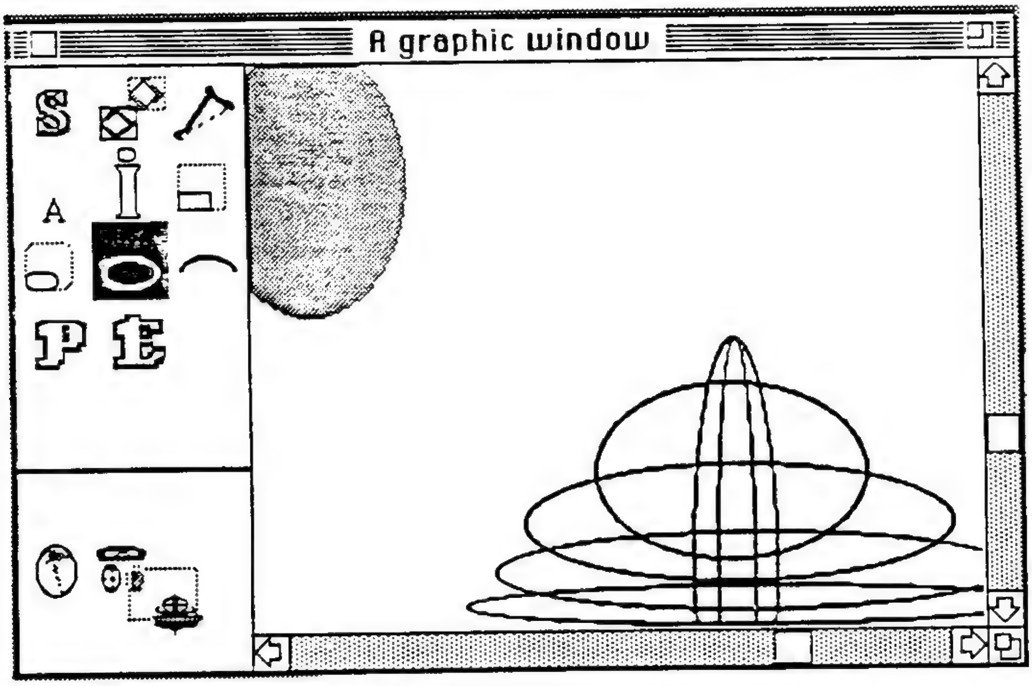
window	: atom, name of graphic window
switch	: atom or variable

USES

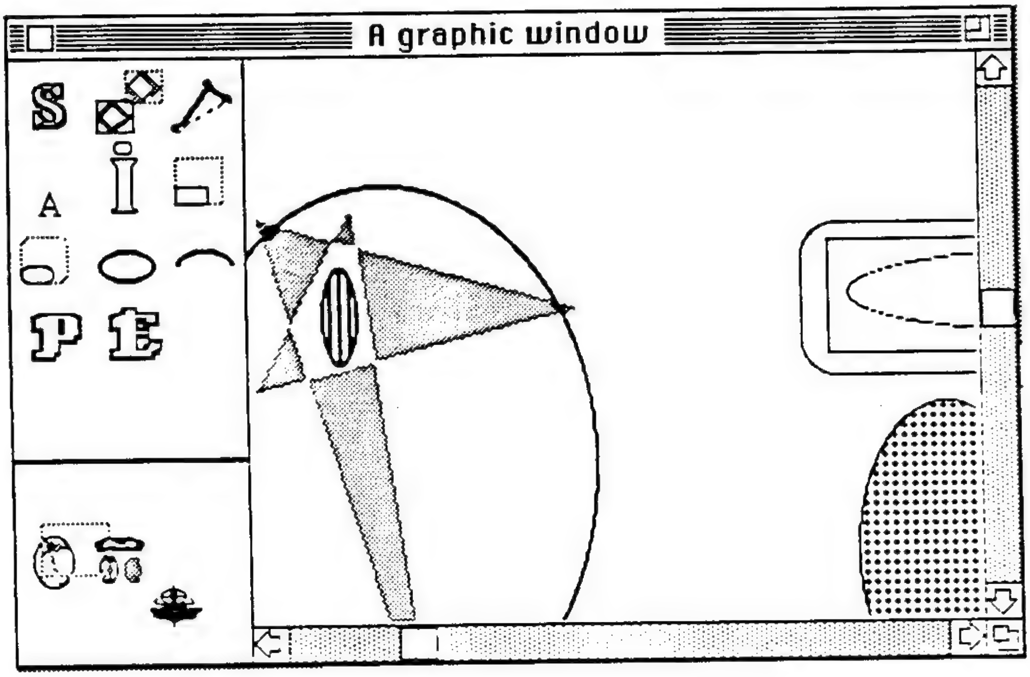
To set or retrieve the viewer setting for the named graphic window.

If *switch* is a variable it will be instantiated to either `on` or `off`.
If *switch* is one of the atoms `on` or `off`, the viewer for the graphic window is turned `on` or `off`.

Turning the viewer on allows you to see a scaled down view of the whole drawing area in the bottom left hand corner of the tool pane. The current location of the viewing pane is represented by a grey rectangle. By dragging this rectangle it is possible to scroll the viewing pane very quickly over the drawing area.



The diagram below represents the above graphic window after dragging the dotted viewer rectangle up and to the left within the viewer area of the tool pane, so causing a 'fast scroll' of the viewing pane over the drawing area of the graphic window.



Note that the size of the viewer depends on the width of the tool pane. The viewer's width is the same as that of the tool pane; its depth is then in the same proportion to the drawing area's depth as its width is to the drawing area's width.

26.3 gmax - the maximum size of a graphic window drawing area

gmax(*window*,*maxy*,*maxx*)

ARGUMENTS

window	: atom, name of graphic window
maxy	: integer or variable, maximum vertical coordinate
maxx	: integer or variable, maximum horizontal coordinate

USE

To set or retrieve the size of the drawing area of the named graphic window .

On creation, a graphic window has a designated drawing area. This is the section of the QuickDraw coordinate plane that can be made visible by scrolling the viewing pane of the window. Pictures that lie outside of this drawing area can be added to the window, but they cannot be made visible by scrolling. By increasing the size of the drawing area, it is possible to bring them into view.

If *maxy* and *maxx* are given, then the size of the drawing area of the named graphic window is changed to the coordinate range

$-maxy \leq Y \leq maxy, \quad -maxx \leq X \leq maxx.$

If *maxy* and *maxx* are both variables, then they are bound to the current values of the maximum Y and X coordinates for the drawing area.

26.4 gsplit - get/set the split for a graphic window

gsplit(*window*,*split*)

ARGUMENTS

window	: atom, name of graphic window
split	: variable or integer, value of split

USES

To set or retrieve the value of the *split* (i.e. the width of the tool pane) in a graphic window.

If *split* is an integer, then the tool pane width for the named graphic window is set to *split* , and the window is redrawn at the next system refresh.

If *split* is a variable, then it will be bound to the current tool pane width for the named graphic window.

26.5 `gview_pane` - get the viewing pane for a graphic window

`gview_pane(`*`window,box`*`)`

ARGUMENTS

window	: atom, name of graphic window
box	: variable, will be bound to a term of the form `box(`*`T,L,D,W`*`)`

USE

To retrieve the location of the viewing pane in the drawing area for a graphic window. The *box* argument is unified with a term of the form

`box(`*`T,L,D,W`*`)`

which describes the current size and position of the viewing pane rectangle over the drawing area of the window.

26.6 `gscroll_to` - scroll the viewing pane to a position

`gscroll_to(`*`window,top,left`*`)`

ARGUMENTS

window	: atom, name of graphic window
top	: integer, vertical coordinate of top of viewing pane
left	: integer, horizontal coordinate of left of viewing pane

USE

To scroll the viewing pane over the drawing area such that the point (*top, left*) becomes the top left hand corner of the viewing pane. The scroll bars and the viewer (if it is on) are redrawn to reflect the new location of the viewing pane.

26.7 `gscroll_by` - scroll the viewing pane by an amount

`gscroll_by(`*`window,down,across`*`)`

ARGUMENTS

window	: atom, name of graphic window
down	: integer, vertical shift
across	: integer, horizontal shift

USE

To scroll the viewing pane over the drawing area by *down* and *across*. The scroll bars and the viewer (if it is on) are redrawn to reflect the new location of the viewing pane.

Window Activation / Deactivation

A window is **activated** when it is made the front window by a mouse click or through a program call to `wfront`. A window is **deactivated** when it was the front window, but another window or dialogue is activated.

Associated with every graphic window is an optional MacPROLOG program that is called every time that window is activated or deactivated. This is referred to as the actdeact program. (This program is associated with the window internally, so even if the window is renamed, the program remains associated with that window.)

26.8 `gactdeact` - activation/ deactivation program

`gactdeact(`*`window,program`*`)`

ARGUMENTS

`window`	: atom, name of graphic window
`program`	: variable or atom, name of program

USES

To set or retrieve the program associated with a graphic window's activation and deactivation (see discussion below).
If *`program`* is given, then the actdeact program for the named graphic window is set to be that *`program`*.
If *`program`* is a variable, then it will be instantiated to the name of the actdeact program for the named graphic window.

The actdeact program must have two clauses. The first clause deals with activation, and the head must be of the form

`program``(activate,`*`Window`*`)`

where *`Window`* is a variable which at the time of the call to *`program`* will hold the name of the window being activated.
The second clause deals with deactivation and the head must be of the form

`program``(deactivate,`*`Window`*`)`

Again, *`Window`* is a variable which will hold the name of the window being deactivated.

The reason that we use a variable as a second argument to hold the name of the window is that although a graphic window can have only one actdeact program associated with it, this program can be associated with more than one graphic window.

WARNING Care must be taken to ensure that the actdeact program does not itself activate or deactivate any other window (for example by generating a dialogue), as this may cause endless recursion.

Example

Given a graphic window called mywindow, we can have the following call to set up an activation / deactivation program called managemenu,

```
gactdeact(mywindow,managemenu)
```

where managemenu is defined by

```
managemenu(activate,Window):-
      install_menu('Margin',['10','20','30','40','50']),
      get_prop(Window,'MARGIN',LastMargin),
      mark_item('Margin',LastMargin).

managemenu(deactivate,Window):-
      kill_menu('Margin').
```

The result of this program is that every time the window mywindow is made active, a menu named **Margin** is installed in the menu bar, and when this window is deactivated, the menu disappears. When the menu is installed it looks for some property (which will have been set up previously) to see which item should initially be marked.

26. 9 Standard Tools

MacPROLOG provides a library of standard tool programs that can be used to select, drag, edit or remove any picture in a window. We recommend that you start by just using these standard tools in graphic windows before developing your own tools.

The tool names given below are the names of the MacPROLOG programs which perform the function of each tool. You may define tools using these names and the predefined programs will automatically be called for those tools. However, the GDL descriptions (representing the graphical appearance of the tools on the screen) are fairly arbitrary and you may replace them with your own picture descriptions if you wish without affecting the behaviour of the tool.

The graphical descriptions given below are predefined MacPROLOG picture descriptions which take no arguments. They can be used as elementary picture descriptors in any GDL description.

`arrow` `info_icon` `rubber` `dragger` `diamond`

The standard tools and their recommended GDL descriptions are as follows.

26.9.1 NAME: `select`
GDL DESCRIPTION: `arrow()`

Allows the user to select one or more pictures by either

a) clicking or shift-clicking over a picture. Clicking makes the picture the only selected picture. Shift clicking adds the picture to the number of selected pictures. To successfully click on a picture, you must click inside what the Mac regards as the picture's interior. In the case of aggregate pictures with overlapping boundaries this might not be what you expect. If you find you can't select the picture by clicking you can always drag a marqui.

b) dragging a rectangle (marqui) in the viewing pane. All pictures <u>completely</u> enclosed by the marqui become selected.

26.9.2 NAME: `drag`
GDL DESCRIPTION: `dragger()`

Allows the user to drag all the currently selected pictures by clicking the mouse over one of these pictures and dragging the mouse.
By clicking the mouse over any non-selected picture without any key pressed, the user can deselect all previous selections, and select and drag this new picture.
If the Shift key is held down whilst clicking on a picture, this toggles the selection of the picture, and the user may drag all the pictures now selected.
If the Option key is held down, then only the picture clicked on is dragged, BUT this does not affect the selected list.
If the Command key is held down then the click will deselect all the pictures.

26.9.3 NAME : `pic_info`
GDL DESCRIPTION: `info_icon()`

Allows the user to get information about a picture by clicking on it. It generates an editable modeless dialogue containing the picture's description, current origin, name, and a scaled version of the picture's graphical image. The picture can be changed by editing its description. It can be shifted by editing its origin or renamed by editing its name. It can also be removed from the window by selecting the **delete** button of the dialogue. An example of this dialogue is displayed at the end of the chapter on Picture Manipulation.

26.9.4 NAME : `enter_text`
GDL DESCRIPTION: `text('Courier',12,0,28,10,'A')`

Allows the user to enter text interactively into an edit field in one of two ways.

a) By clicking and immediately releasing the mouse in the viewing pane. The user sets up an expanding editable line of text, into which all subsequent keyboard input will be directed until a click occurs outside this line of text.

b) By dragging a marqui in the viewing pane. The user sets up a 'fixed size' edit rectangle into which all subsequent keyboard input will be directed until a click occurs outside this rectangle.

In both cases the active edit field created in the graphic window behaves in the same way as one in a dialogue, but with the extra ability to switch fonts, font style, and font size.

A click outside the active edit field, but still inside the graphic window, will deactivate and close the edit field. At this point the system takes the text from the edit field and converts the information into a GDL description using either the `text` or `textbox` picture descriptors. It then generates a name for this new picture and adds it to the window's picture list.

The `enter_text` tool also allows these pictures of text to be re-edited by selecting the tool and clicking in one of these pictures. This opens an editable line or box of text (depending on whether it is `text` or `textbox`), and initialises the edit field with the text associated with the picture.

26.9.5 NAME : `eraser`
GDL DESCRIPTION: `rubber()`

When the `eraser` tool is selected, a click over a picture will delete the picture from the window.

NOTES

i) A double click on any of the above tools will generate an 'About Tool' dialogue.
ii) See the `add_tools` primitive below for how to add these tools to a graphic window.
iii) See the Tool Building chapter for further details on tool programs.

26.10 `add_tools` - add tools to a graphic window

`add_tools(`*`window, tools`*`)`
`add_tools(`*`window, tools, columns`*`)`

ARGUMENTS

window	: atom, name of window
tools	: list of terms describing tools
columns	: integer, number of display columns for tools

USES

Each tool on the `tools` list is described by a compound term of the form

name(description)

where *name* is the name of the tool - i.e. the name of the program that will be called when the tool is used - and *description* is a GDL description of the tool's picture in the tool pane.

1. Two argument use: Each of the tools in `tools` is added to the list of existing tools for the *window*. The number of tool display columns remains unaltered, and the additional tools are drawn starting at the next available rectangle in the tool pane. (See discussion below).

2. Three argument use: The number of tool display columns is also given. If this has been changed since the last `add_tools` call for that window, then the set of tools is reconfigured on the screen.

The first time `add_tools` is called, if the number of tool columns if not given it is set to 2. If there are too many tools to fit in the tool pane then only those that fit, or partially fit, will be drawn.

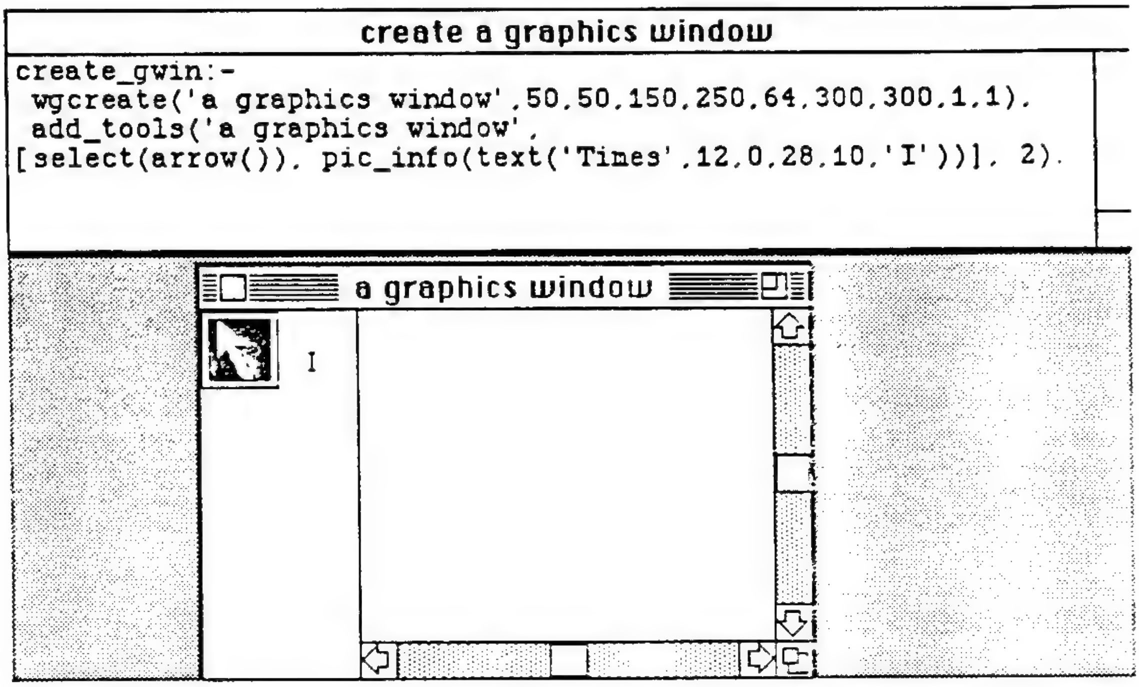
Tool pictures must be described within a 32x32 pixel rectangle. Anything defined outside this rectangle will not be visible. The tool pictures are drawn in the tool pane from left to right (with the specified number per row), and from top to bottom. The graphics system calculates the height and width for each displayed tool picture by dividing the width of the tool pane by the number of tool columns. It then scales each tool accordingly.

The width of the tool pane (the *split*) is adjustable. When the tool pane is assigned zero width, no tools will be visible, but they still exist. However, as the user cannot see any tools, it is impossible to select any by clicking the mouse. The tool selection can still be changed by a call to `set_tool`, or through the **Window details...** dialogue.

Each graphic window has associated with it a graphical description of its **current cursor**. This cursor is displayed whenever the mouse is in the drawing pane of the window. Individual tools can modify the appearance of the cursor using the `gcursor` primitive.
For example, a tool that creates a new picture object through some interactive process like dragging a marqui or a rubber band might change the appearance of the current cursor to a pen, whereas a text-based tool, like `enter_text`, might change it to the I-beam. The standard tools above change the cursor in this way.

Example

The following graphic window with 2 tools is created by a call to the `create_gwin` program.



In this example, the `select` tool is represented by the predefined picture `arrow()`, and `pic_info` is represented by the letter I.

26.11 `del_tools` - remove tools from a graphic window

```
del_tools(window)
del_tools(window,toolnames)
del_tools(window,toolnames,columns)
```

ARGUMENTS

window	: atom, name of window
toolnames	: list of atoms that are names of tools in the window
columns	: integer, number of columns

USES

1. One argument use: All the tools are removed from the named graphic window.

2. Two argument use: The named tools are removed from the list of existing tools for the named graphic window. The number of columns remains unaltered, and the new, smaller, set of tools is reconfigured and drawn in the tool pane at the next system refresh.

3. Three argument use: The number of tool display columns is also given. The named tools are removed from the list of existing tools for the graphic window, and the new, smaller, set of tools is reconfigured and drawn in the tool pane at the next system refresh, with the new number of columns.

26.12 `get_tools` - get names of tools in a graphic window

```
get_tools(window,toolnames)
```

ARGUMENTS

window	: atom, name of graphic window
toolnames	: list of atoms that are names of tools in the window

USE

To retrieve the list of names of the tools associated with the *window*.

26.13 `tool_desc` - get the GDL picture description of a tool

```
tool_desc(window,name,desc)
```

ARGUMENTS

window	: atom, name of new window
name	: atom, name of a tool in the window
desc	: variable, will be bound to the GDL description of the tool

USE

To retrieve the description of a named tool in a graphic window. After the call *desc* will be instantiated to the term describing the named tool.

26.14 `gcols` - the number of columns of the tool pane

`gcols(`*`window, cols`*`)`

ARGUMENTS

`window`	: atom, name of graphic window
`cols`	: integer or variable

USES

To set or retrieve the number of columns in the tool pane of the named graphic window .

If *`cols`* is an integer, then the number of tool display columns is set to *`cols`* . The tool pane is reconfigured, and the tools redrawn at the next system refresh.

If *`cols`* is a variable, then it will be bound to the current number of tool display columns.

26.15 `get_tool` - get the current tool

`get_tool(`*`window, tool`*`)`

ARGUMENTS

`window`	: atom, name of graphic window
`tool`	: variable, will be bound to name of current tool

USES

To get the name of the currently selected tool in the graphic window. If there are no tools in the window, the call will succeed with *`tool`* bound to `true`.

26.16 `set_tool` - set the current tool

`set_tool(`*`window, tool`*`)`

ARGUMENTS

`window`	: atom, name of graphic window
`tool`	: atom, name of tool

USES

To set the currently selected tool in the named graphic window .

If *`tool`* is the name of a tool, then the current tool is set to this tool. The tool pane is updated to reflect this change: the old tool is deselected and the new tool selected. Where the old tool and new tool are different, any activation/ deactivation programs associated with the tools will be invoked.

27 Advanced Drawing

In this chapter we consider the mechanics of when and how the Mac refreshes a graphic window. This will introduce the concepts of transient pictures, 'fast' drawing, invalidation, validation and refreshing.

We will also look at how resources can be used inside the MacPROLOG graphics system.

Refreshing a Graphic Window

There are two ways in which the viewing pane of a graphic window gets updated.

It is updated when a picture is drawn in the viewing pane, say by an `add_pic` call. Drawing in a window results in an immediate update of the viewing pane of the window.

It is also updated when the Macintosh refreshes its display, and certain areas of the viewing pane have been **invalidated**. An area of screen can be invalidated when it no longer correctly reflects the logical structure of that part of the screen. For example `del_pic` invalidates the area of the viewing pane that was occupied by the removed picture. However, the redrawing takes place only when the MacPROLOG system next relinquishes control to the Macintosh system, for example at the end of an evaluation or when a modeless dialogue is displayed. This is why `del_pic` does not immediately result in a redrawing of the viewing pane.

Certain activities automatically invalidate parts of the viewing pane. When a window is made visible, resized or zoomed, then all of the window will be invalidated. But covering and uncovering with another window will invalidate only those parts that were covered but are now visible.

When parts of a graphic window's viewing pane need to be refreshed, the list of picture descriptions recorded for that window is accessed and used to redraw the viewing pane.

What `add_pic` does

`add_pic` does two things. Firstly, if the picture lies in the viewing pane of the window, it immediately draws the picture in the viewing pane. Secondly, it records the fact that the picture has been drawn in the window by adding it to the picture list for the window.

But `add_pic` might have done something different.

It could have added the picture to the picture list for the window and instead of immediately drawing the picture in the viewing pane, it could have simply invalidated the area of the viewing pane that the picture would have occupied. The picture would then be drawn when the Macintosh next refreshed the screen.

Alternatively, it could have just added the picture to the picture list for the window. The picture would then appear only when the viewing pane was next invalidated, for example by resizing or zooming the window, but not before.

In either case, the long term effect is the same, i.e. a new permanent picture is associated with the window. The picture is permanent in the sense that whenever the window is refreshed, the picture is redrawn if it lies within the viewing pane of the window.

There is one other thing that `add_pic` might do. It could just draw the picture and not add its description to the picture list for the window. The effect of this would be to make the picture a transient picture of the window, meaning that the next time the window is refreshed the picture will probably disappear because the redraw program does not 'know' about the picture.

There is a role for transient pictures. Tools for building pictures may need to draw transient pictures: for example a tool for constructing an object as a series of closed lines. Only at the end of the tool program is it appropriate to record the description of the drawn picture in the window's picture list to make the picture permanent.

Tools that generate dynamic and fast moving images also need to draw transient pictures, for example an "Eight Queens" program. Only the final state of the picture need be recorded, when all the changes have been made.

The primitives in this chapter allow the programmer to

a) immediately draw a picture in a window without updating the window's list of pictures
b) ensure the refreshing of certain parts of a window at the next system refresh, by invalidation.
c) inhibit the refreshing of certain parts of a window at the next system refresh, by validation.
d) force an immediate refresh of all invalidated areas of a window
e) update the window's list of pictures without causing a redraw

Explicit drawing

27.1 **draw_pic - draw a transient picture**

```
draw_pic(desc)
draw_pic(window, desc)
draw_pic(window, desc, org)
```

ARGUMENTS

window	: atom, name of graphic window
desc	: a (possibly extended) picture description in GDL
org	: a point pair (*Y*, *X*), the local origin

USE

To immediately or 'fast' draw a transient picture in a graphic window without recording the picture in the window's picture list.

1. Three argument use
Fast draw the description in the named graphic window, using the given local origin.

2. Two argument use
Fast draw the picture in the named graphic window, with a local origin of (0,0).

3. One argument use
Fast draw the description in the **Default Graphic Window** with a local origin of (0,0).

In a graphical version of the "Eight Queens" problem, a queen can be visually removed from a square by a call of the form

```
draw_pic(board,blankbox,whitepen(white(fillbox(D,A,32,32))))
```

where D and A identify the chessboard square.

NOTE
`draw_pic` does not update the window's picture list. If you want the drawn picture to be a *permanent* picture of the window, i.e. to be redrawn when the window is refreshed, you should either use `add_pic`, or, in addition to the `draw_pic` call do an explicit `record_pic` on the window. Use `draw_pic` for fast updating of dynamic pictures that will change several times during a given query (such as a graphic trace of a program solving the "Eight Queens" problem or "Tower of Hanoi"). Just before the query terminates you can record the final state of the picture using `record_pic` to update the picture list for the window.

Invalidation

Invalidation is used in connection with explicit drawing.

Suppose that you have drawn some transient pictures in a window on top of some permanent pictures. You now wish to remove the transient pictures and make the permanent pictures underneath visible again. For this you would use invalidation.

Minimally, you need to invalidate the areas occupied by the transient pictures. The next time the Mac refreshes its display, the permanent pictures will reappear. If you want the permanent pictures to immediately reappear, you need to invalidate the relevant areas and force a refresh of the display using the `refresh_now` primitive (see below). This refreshes all the currently invalidated areas of the window and sets them back to valid.

You can invalidate the whole of the viewing pane of a graphic window, or just part of it. For faster refreshing it is best to invalidate the smallest area that encloses the picture to be redrawn. This area is referred to as the picture's frame, and is calculated and recorded by the system when a picture's description is first added to a window's picture list.

`del_pic`, `shift_pic` and `edit_pic` all use invalidation to make the parts of the screen underneath the moved or removed picture be redrawn. They also invalidate the viewer if it is present. The effect of using these primitives is therefore only visible at the next system refresh.

Validation

Validation is the converse of invalidation. It causes the Macintosh to remove invalidated areas from its records, and this will prevent a redraw of those parts of a window. This can be useful to ensure that certain parts of the screen image are not redrawn at the next system refresh.

Validating the Viewer

A call to `refresh_now` will also redraw the viewer for the window, which can be quite a time consuming operation. If you are making several changes to a window by shifting or editing pictures and you want to repeatedly call `refresh_now` to make the changes immediately visible, it is better to delay the updating of the viewer until the end of the sequence of changes. The refresh of the viewer can be inhibited by validating it before the call to `refresh_now` (see the `val_viewer` primitive). At the end of the sequence of changes you could then call `refresh_now` without validating the viewer, so that it too will be updated to reflect the final state of the window.

27.2 `inval_pic` - invalidate the 'frame' of a picture

```
inval_pic(window,desc)
inval_pic(window,desc,org)
```

ARGUMENTS

window	: an atom, name of graphic window
desc	: a (possibly extended) picture description, as in `draw_pic`
org	: a point pair `(Y,X)`, the local origin

USE

To invalidate the rectangle that would be occupied by the described picture. This may be used for instance to redisplay what lies underneath a transient picture drawn by `draw_pic`.

27.3 `inval_box` - invalidate a given rectangle

```
inval_box(window,box)
inval_box(window,box,org)
```

ARGUMENTS

window	: an atom, name of graphic window
box	: a term describing a rectangle
org	: a point pair `(Y,X)`, the local origin

USE

To invalidate a given rectangle. The *box* argument must be a term of the form

```
box(T,L,D,W)
```

For example, we could write a program to invalidate the viewing pane of a graphic window as follows.

```
inval_viewing_pane(Window):-
      gview_pane(Window,Rect),
      inval_box(Window,Rect).
```

27.4 `inval_tool_pane` - invalidate the tool pane

`inval_tool_pane(`*`window`*`)`

ARGUMENT

`window` : atom, name of graphic window

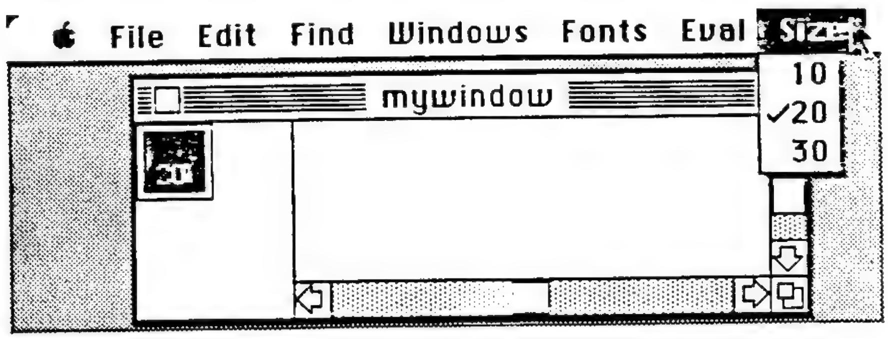
USE

To invalidate the tool pane of a given graphic window. If we had a tool picture whose representation changed depending on, say, a menu selection, then we could invalidate the tool pane whenever that menu was changed, thus forcing the up-to-date description of the tool to be drawn.

Example

In the program below, we have a tool `sizetool` in the tool pane whose description is a user form, `newsize`, which is sensitive to a property associated with a menu **Size**. The graphical representation of the tool is the number representing the currently selected text size. Therefore, whenever the selected menu item is changed we need to update the tool's appearance in the tool pane by a call to `inval_tool_pane`.

```
go:-                                                   % initialisation routine
   wgcreate(mywindow,50,50,200,200,64,300,500,1,1),
   add_tools(mywindow,[sizetool(newsize(mywindow))]),
   install_menu('Size',['10','20','30']),              % install menu
   mark_item('Size','10'),                             % initially mark an item
   set_prop(mywindow,'SIZE','10').                     % and set a property

'Size'(Value):-                                        % deal with menu selection
   on(Value,['10','20','30']),                         % check its a valid size
   get_prop(mywindow,'SIZE',Previous),                 % get previous value
   Previous =\= Value,                                 % if different menu selection
   unmark_item('Size',Previous),                       % update menu marks
   mark_item('Size',Value),                            % mark the new selection
   set_prop(mywindow,'SIZE',Value),                    % update property
   inval_tool_pane(mywindow).                          % redraw tool pane

newsize(Window,Desc):-                                 % user defined descriptor
   get_prop(Window,'SIZE',Size),                       % get current size to be shown
   Desc = text('Times',12,0,28,6,Size).                % return new description
```



27.5 `inval_viewer` - invalidate the viewer

`inval_viewer(`*`window`*`)`

ARGUMENT

window	: atom, name of graphic window

USE

To invalidate the viewer for a given graphic window if it is on. If the viewer is not on, the call succeeds but has no visual effect.

27.6 `val_box` - validate a rectangle

`val_box(`*`window, box`*`)`
`val_box(`*`window, box, org`*`)`

ARGUMENTS

window	: an atom, name of graphic window
box	: a term of the form `box(`*`T, L, D, W`*`)`
org	: a point pair of the form `(`*`Y, X`*`)`, the local origin

USE

To validate a given rectangle, and thus inhibit a refresh of that part of a window.

27.7 `val_viewer` - validate the viewer

`val_viewer(`*`window`*`)`

ARGUMENT

window	: atom, name of window

USE

To validate the viewer for a given graphic window, if it is on. If the viewer is not on, the call succeeds but has no visual effect.

27.8 `val_pic` - validate the 'frame' of a picture

```
val_pic(window, desc)
val_pic(window, desc, org)
```

ARGUMENTS

window	: an atom, name of graphic window
desc	: a (possibly extended) picture description, as in `draw_pic`
org	: a point pair `(Y, X)`, the local origin

USE

To validate the rectangle that would be occupied by the described picture.

27.9 `refresh_now` - refresh a window

```
refresh_now (window)
```

ARGUMENT

window	: an atom, name of graphic window

USE

To force the immediate redrawing of all the invalidated areas of a graphic window. To be used when you do not want to wait for the system to refresh a window.

EXAMPLE

There are numerous ways of writing a program that translates a given list of pictures. We compare and contrast the different graphical behaviours of three similar but different programs.

In the program below we backtrack through a list of pictures, shifting each one, and at the end we force a redrawing of the window in which the pictures lie. Nothing will visibly happen before then.

```
move_pics(Window,Pics):-
    on(Pic,Pics),                          % for all pictures on the list
    shift_pic(Window,Pic),                 % shift one
    fail.                                  % backtrack to on

move_pics(Window,Pics):-
    refresh_now(Window).                   % force a redraw
```

In the next program we again backtrack through the list of pictures, shifting each one, but force a redraw of the window after every shift.

```
move_pics(Window,Pics):-
    on(Pic,Pics),                          % for all pictures on the list
    shift_pic(Window,Pic),                 % shift one
    refresh_now(Window),                   % force a redraw
    fail.                                  % backtrack to on

move_pics(Window,Pics).
```

However, if the viewer is on for the window, then the viewer will also be redrawn after each shift. This could be quite slow for a long picture list. We could inhibit the viewer being refreshed by a call to `val_viewer`, as below.

```
move_pics(Window,Pics):-
    on(Pic,Pics),                          % forall pictures on the list
    shift_pic(Window,Pic),                 % shift one
    val_viewer(Window),                    % inhibit refresh of viewer
    refresh_now(Window),                   % force a redraw of window
    fail.                                  % backtrack to on

move_pics(Window,Pics):-
    inval_viewer(Window),                  % force refresh of viewer
    refresh_now(Window).                   % force a redraw .
```

27.10 **`record_pic`** - add a picture to a window without displaying it

```
record_pic(desc)
record_pic(name, desc)
record_pic(window, name, desc)
record_pic(window, name, desc, origin)
```

ARGUMENTS

desc	: a (possibly extended) picture description in GDL
name	: an atom, name of picture to be inserted
window	: an atom, name of graphic window
origin	: point pair, amount of local shift

USE

To record the given picture in the picture list for a graphic window but not immediately draw it. (The drawing will take place at the next system screen refresh.)

1. One argument use
Generates a unique name for the picture, inserts it into the **Default Graphic Window**, and gives it a local origin of (0,0)

2. Two argument use
Inserts the named picture into the **Default Graphic Window**, and gives it a local origin of (0,0)

3. Three argument use
Inserts the named picture into the named window, and gives it a local origin of (0,0)

4. Four argument use
Inserts the named picture into the named graphic window with the given local origin.

NOTE A `draw_pic` followed by a `record_pic` is equivalent to an `add_pic`. See the discussion at the beginning of this chapter.

27.11 `record_qpic` - add a picture in QuickDraw format without displaying it

```
record_qpic(desc)
record_qpic(name,desc)
record_qpic(window,name,desc)
record_qpic(window,name,desc,origin)
```

ARGUMENTS

desc	: a (possibly extended) picture description in GDL
name	: an atom, name of picture to be inserted
window	: an atom, name of graphic window
origin	: point pair, amount of local shift

USE

Exactly the same as `record_pic`, except that the picture will be added as a QuickDraw picture. As with the corresponding `add_qpic` call, this speeds up subsequent redrawing of the picture, but you lose the picture's Prolog structure.

27.12 `remove_pic` - 'silent' removal of a picture

```
remove_pic(window,name)
```

ARGUMENTS

window	: atom, name of a graphic window
name	: atom, name of a picture

USE

This is similar to `record_pic` in that no immediate refreshing of the screen takes place. The named picture is removed from the window's list of pictures, but it will remain visible on the screen until the screen is next updated.

Using Resources

MacPROLOG Graphics provides primitives to access resources of type `'PICT'`, `'CURS'` or `'ICON'`.

Resource pictures can be accessed using the GDL `picture` descriptor, and saved to disk using `save_pic`.
Resource icons can be accessed using the GDL `icon` descriptor.
Resource cursors can be accessed using `gcursor`.

EXAMPLE 1

The following program displays in a graphic window all the icons in currently open resource files. (The `res_items` primitives is documented in the Resources chapter.)

```
add_all_icons(Window):-
    res_items('ICON',ListOfIcons),          % get list of resource icons
    add_icons(Window,ListOfIcons,0).        % start with zero

add_icons(Window,[],N).
add_icons(Window,[H|Rest],N):-             % work through list of icons
    add_one_icon(Window,H,N),               % add the head of the list
    N1 is N+1,                              % increase counter
    add_icons(Window,Rest,N1).              % work through the rest

add_one_icon(Window,[RName,RNum,File],N),  % add Nth icon
    gensym(resource_icon,NextIcon),         % generate a new name
    NextX is N**32,                         % move along 32 pixels
    add_pic(Window,NextIcon,icon(0,NextX,32,32,RNum)).
                                            % add it to the window
```

EXAMPLE 2

The following program that finds all the available resource pictures in currently open resource files. and then adds them to a graphic window. using their original dimensions.

```
add_all_pics(Window):-
    res_items('PICT',ListOfPics),           % Get list of resource pictures
    on([Name,Num,File],ListOfPics),         % Find a picture on the list
    gensym(resource_picture,NextPict),      % Generate a new name
    pict_frame(resource(File,Num),box(T,L,D,W)), % Get its dimensions
    add_pic(Window,NextPict,picture(T,L,D,W,resource(File,Num))),
    fail.                                   % Add it to the window and fail
                                            %   (This backtracks to on)
add_all_pics(Window).                       % Succeed when there are no
                                            % more pictures
```

27.13 **save_pic** - save a picture in a resource file

save_pic(*name*, *desc*, *file*)
save_pic(*name*, *desc*, *file*, *volume*)

ARGUMENTS

name	: atom, name of resource picture
desc	: a (possibly extended) picture description in GDL
file	: atom, name of resource file
volume	: integer, volume identifier

USE

To save a picture description to disk as a permanent resource picture.

1. Three argument use The default volume is used.

2. Four argument use: The picture is saved in the named *file* on the given *volume*.
file can be the name of an existing MacPROLOG program file, or it can be a new file name.
If the file already exists and has a resource fork, then the resource fork will be opened and the picture saved there.
If the file already exists but has no resource fork, then the resource fork will automatically be created.
If the file does not exist, then both the file and the resource fork will automatically be created.

Once a picture exists on disk as a resource, it can be displayed in either a graphic window or a dialogue very efficiently using a picture description of the form

`picture(T,L,D,W,resource(Name,File))`

or

`picture(T,L,D,W,resource(Name,File,Volume)).`

27.14 `qpic_frame` - return a QuickDraw picture's dimensions

`qpic_frame(`*`picture_details`*`,`*`rectangle`*`)`

ARGUMENTS

`picture_details` : description of a resource picture
`rectangle` : variable, will be unified with a term of the form `box(`*`T,L,D,W`*`)`

USE

To find out the dimensions of a QuickDraw picture. This can be useful before using the picture in a picture description. Getting and using the original dimensions prevents distortion of the picture.

Examples of valid calls:

```
qpic_frame(clipboard('1'),Rectangle)
qpic_frame(resource(Name,File),Rectangle)
qpic_frame(resource(Name,File,Volume),Rectangle)
```

Example
The following program will add a QuickDraw resource picture to a graphic window in its original dimensions and at its original coordinates.

```
add_qpic(Window,Name,File) :-                              % add a QuickDraw picture
     qpic_frame(resource(Name,File),box(T,L,D,W)),  % get dimensions
     gensym(qpic,PicName),                                % generate a name
     add_pic(Window,PicName,picture(T,L,D,W,resource(Name,File))).
                                                           % add it to window
```

27.15 `qpic_size` - return a QuickDraw picture's size

`qpic_size(`*`picture_details`*`,`*`numberofbytes`*`)`

ARGUMENTS

`picture_details` : description of resource picture
`numberofbytes` : variable. will be bound to size of picture

USE

To find out the size of a QuickDraw picture. Useful in memory management. You may find that larger pictures can occupy up to 12K or more of memory.

Example uses:

```
qpic_size(clipboard('1'),Numberofbytes)
qpic_size(resource(Name,File),Numberofbytes)
qpic_size(resource(Name,File,Volume),Numberofbytes)
```

27.16 **`gcursor` - associate a new cursor with the viewing pane**

`gcursor(`*`window, cursor`*`)`

ARGUMENTS

window	: atom, name of graphic window
cursor	: a cursor descriptor as listed below

USE

To change the appearance of the current cursor associated with the viewing pane of a graphic window.

A *cursor* descriptor is a term with one of the following forms

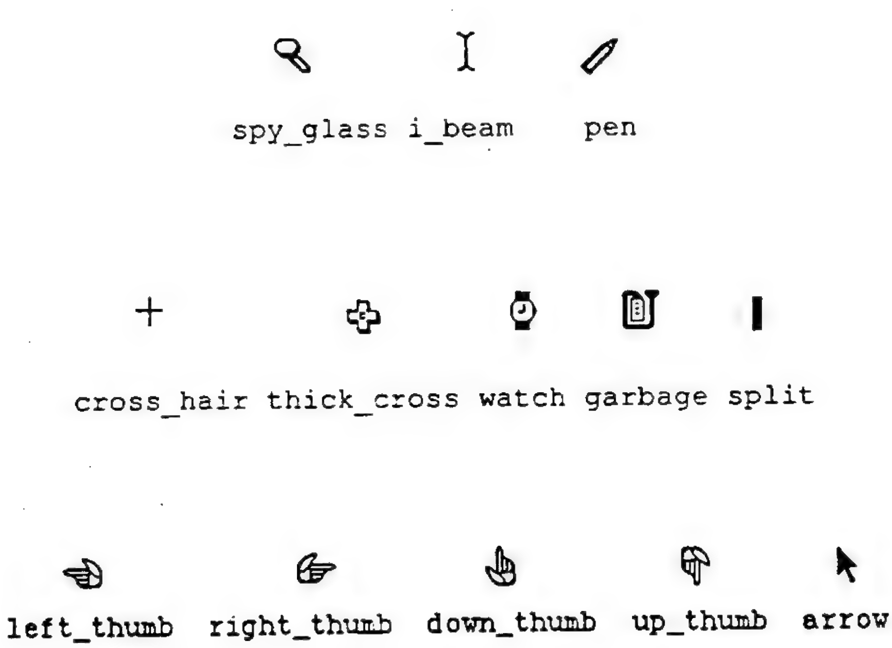
a) `resource(`*`num, file, vol`*`)`

b) `resource(`*`num, file`*`)`

c) `resource(`*`num`*`)` for a currently open resource file

d) `user(`*`hex_atom`*`)`
where *hex_atom* is an atom comprised of 128 hexadecimal characters

e) one of the following reserved cursor names which denote the displayed cursors



28 Tool Building

In the chapter on Graphic Windows we described some standard tools that can be used in graphic windows. In this chapter we show how you can build your own specialized tools. We describe the required format of tool programs and a number of primitives which can be used to help in implementing specialized tools.

The format for tool programs

Recall that a tool in a graphic window is implemented by a PROLOG program, and its picture in the tool palette is determined by a GDL description. The name given to the tool in the graphic window is also the name of the PROLOG program invoked whenever the user clicks in the relevant parts of the window.

In fact there are a number of circumstances in which a tool program is invoked: when the tool is first selected from the tool palette, when the user clicks in the viewing pane, and when the tool is deselected (before another tool is selected from the tool palette). The forms of these invocations are:

	Form of call	Trigger for call
1.	`tool(activate,Window)`	tool is selected
2.	`tool(Window,Y,X,Mod)`	mouse click in drawing pane
3.	`tool(deactivate,Window)`	tool is deselected
4.	`tool(double,Window)`	double click occurs on the tool picture
5.	`tool(close_edit,Window)`	tool has opened an edit field which has been closed by a click outside the field

1. A tool is selected by the user clicking on the picture representing it in the tool palette. It becomes the currently selected tool, and is activated. The **activate** mode of the tool is invoked. The format of this call is

```
tool(activate,Window)
```

where `activate` is an atom which denotes the activate mode, and *Window* will be the name of the graphic window containing the tool. The function of the activate mode is to allow the tool to perform any initialisations prior to it being invoked.

A common function for an activation mode is to change the current cursor using the `gcursor` primitive.
For example:

```
mytool(activate,Window):-
      gcursor(Window,pen).
```

will cause the cursor to be switched to a pen when `mytool` is activated. This cursor will be displayed whenever the mouse is moved into the viewing pane of the graphics window. Changing the cursor can give the user a visual indication of the nature of the tool to be invoked.

2. When the mouse is clicked in the viewing pane, MacPROLOG invokes the currently selected tool program with the details of the mouse click as parameters. This is the **click mode** of the tool, usually the most important mode. The format for this call is

```
tool(Window,Y,X,Mod)
```

where `Y` and `X` are the down and across coordinates of the mouse click in the drawing area of *`Window`*, and *`Mod`* is an integer indicating the keyboard status at the time of the click, i.e. whether or not any of the 'modifier' keys such as *Shift* or *Option* were down when the mouse click occurred. The *`Mod`* value can be:

0	No key down
256	Command key down
512	Shift key down
1024	Caps Lock key down
2048	Option key down

This information is important for some tools where the state of the modifier keys affects the behaviour of the tools. For example, traditionally, a shift-click **extends** a selection, whereas a click without any modifiers selects a new object.

3. The third mode is the **deactivate** mode. This is invoked when a tool is deselected, i.e. it was active, but another tool has now been selected. The format for this deactivate call is

```
tool(deactivate,Window)
```

where `deactivate` is an atom which denotes the deactivate mode, and *`Window`* is the name of the graphics window containing the tool. Example:

```
mytool(deactivate,Window):-
      gcursor(Window,cross_hair).
```

This deactivate mode allows the programmer to 'tidy up' after using a tool.

4. The fourth mode is the **double** mode. This is invoked when the user double-clicks on a tool. The format for this call is

```
tool(double,Window)
```

where `double` is an atom which denotes the double mode, and *`Window`* is the name of the graphics window containing the tool.
For example, one might give the user help about a tool if they double click on the tool. This could be done by having a help file (see the `help` primitive) called 'Application Help' (say) and a *`tool`* clause of the form:

```
mytool(double,Window):-
      help('Application Help',tool3,'').
```

5. The fifth mode is the **close_edit** mode. This use for edit fields is explained in more detail below.

Note
The `activate`, `deactivate`, `double` and `close_edit` modes of a tool program are optional - the call can immediately fail due to the absence of a matching clause. However, a tool program *must have* a 'click' mode defined (mode 2 above).

Tool Building Primitives

28.1 **marqui** - interactively draw a marqui

marqui(*window,pivot,marquirect*)

ARGUMENTS

window	: atom, name of graphic window
pivot	: point pair of the form *(Y, X)*, fixed point of marqui
marquirect	: variable, will become the rectangle drawn

USE

`marqui` enables the user to drag a marqui, which is a grey hollow rectangle that follows the mouse while its button is being held down, with one corner fixed at *pivot*. Typically this is where the mouse was clicked in the viewing pane. When the mouse button is released, the rectangle that has been drawn is returned as the value of *marquirect*, which will be a term of the form

box(*T, L, D, W*)

The `marqui` call will fail if the mouse does not move at all (i.e. if the user simply clicks and releases the mouse in the same place).

Example use

You may often use `marqui` in the 'click mode' program of a tool, passing in the coordinates of the mouse click to `marqui`. Below is a tool, which we might call `add_filled_oval`, that enables the user to interactively add a grey filled oval.

```
add_filled_oval(Win,Y,X,Mod):-
    marqui(Win,(Y,X),box(T,L,D,W)),              % get the marqui rectangle
    gensym(filledoval,Name),                     % generate a name
    add_pic(Win,Name,grey(filloval(T,L,D,W))).   % add a grey filled oval
```

Note the use of `gensym` to create a new unique name for the picture element about to be added.

28.2 `find_pic` - find the uppermost picture under the mouse

`find_pic(`*`window,point,name`*`)`
`find_pic(`*`window,point,name,desc,org`*`)`

ARGUMENTS

window	: atom, name of graphics window
point	: point pair of the form `(`*`Y,X`*`)`, position of mouse
name	: variable, will be bound to the topmost picture under *point*
desc	: variable, becomes description of picture found
org	: variable, becomes origin of picture found

USE

Given a mouse position as a point pair, `find_pic` will return the name of the **uppermost** picture found underneath the mouse position, and optionally its description and origin.

For example, a tool called `eraser` that deletes a mouse selected picture from a graphics window, and clears it from the screen on the next system refresh, might have its **click** mode defined as follows.

```
eraser(Window,Y,X,Mod):-
      find_pic(Window,(Y,X),Name),
      del_pic(Window,Name).
```

28.3 `replace_pic` - find and replace the picture under the mouse

`replace_pic(`*`window,point,name,desc,newdesc`*`)`

ARGUMENTS

window	: atom, name of graphics window
point	: point pair of the form `(`*`Y,X`*`)`, position of mouse
name	: variable, will be bound to name of picture underneath *pt*
desc	: variable, will be bound to old picture description
newdesc	: new picture description

USE

This is a 'find and replace' usage which is more efficient than accessing the picture twice, say with `find_pic` and then `chg_pic`. Given a mouse position as a point pair, `replace_pic` will return the **uppermost** picture found underneath the cursor position. The picture will be redrawn using *newdesc*, and its description in the picture list will be replaced as well.

<u>Example use</u>
A tool that scales a mouse selected picture about (0,0) by a factor of 2 in both directions could be defined by

```
scaler(Win,Y,X,Mod):-
      replace_pic(Win,(Y,X),Pic,Desc,scale(2,2,Desc)).
```

28.4 `find_pics` - find the list of pictures under the mouse

`find_pics(`*`window,point,names`*`)`

ARGUMENTS

window	: atom, name of graphics window
point	: point pair of the form (*Y*, *X*), position of mouse
names	: variable, will be bound to list of picture names

USE

Given a mouse position as a point pair, `find_pics` will return the list of all the pictures found underneath this mouse position, where the first in the list is the uppermost of the pictures, and the last the bottommost.

For example, a tool that makes the bottom picture under the mouse the top picture might be defined as

```
bring_to_top(Window,Y,X,Mod):-
        find_pics(Window, (Y,X),Pics),
        get_last(Pics,Name),              % get last name on Pics
        bring_to_front(Window,[Name]).
```

where `get_last` is some program defined to return the last element of a list.

28.5 `pics_in_box` - find the list of pictures in a rectangle

`pics_in_box(`*`window, rectangle, names`*`)`

ARGUMENTS

window	: atom, name of graphics window
rectangle	: term of the form box(*T*, *L*, *D*, *W*)
names	: variable, will be bound to list of picture names

USE

Given a rectangle, `pics_in_box` will return the list of all those pictures that lie completely inside that rectangle. This primitive looks at a picture's frame to see if it lies in the rectangle.

28.6 **`pt_in_pic`** - see if a point is in a picture

`pt_in_pic(`*`window,point,name`*`)`

ARGUMENTS

`window`	: atom, name of graphics window
`point`	: point pair of the form `(`*`Y,X`*`)`
`name`	: atom, name of picture

USE

Given a point and a picture name, `pt_in_pic` succeeds if the point is in the picture, and fails otherwise. The call succeeds if the point is actually in the picture's interior, and not just inside its frame. In the case of overlapping aggregate pictures, and polygons with crossing lines, this interior may be a lot smaller than you'd expect.

28.7 **`rubber_band`** - drag a rubber band

`rubber_band(`*`window,point1,point2`*`)`

ARGUMENTS

`window`	: atom, name of graphics window
`point1`	: point pair of the form `(`*`Y,X`*`)`, initial point
`point2`	: variable, will be bound to a point pair, final point

USE

Given an initial point in the viewing pane, `rubber_band` allows the user to drag a rubber band (i.e. a grey dotted line) around the viewing pane. When the mouse is released, a grey dotted line is drawn connecting the initial point, *`point1`*, to the point where the mouse was finally released, *`point2`*.
(You may then want to call `add_pic` to make this last line permanent.)

Rubber Band Example

A tool called `add_lines` which allows the user to build up a set of joined lines by dragging a rubber-band might have its click mode defined as follows.

```
add_lines(Win,Y,X,Mod):-
    gcursor(Win,pen),                              % make cursor a pen
    rubber_band(Win,(Y,X),Nxt),                    % get first line
    rubber_loop(Win,Nxt,[Nxt,(Y,X)],AllPts,0),     % pass 0 as initial Modifier
    gensym(userLines,Syname),                      % get a unique name
    add_pic(Win,Syname,lines(AllPts)).             % add joined lines picture

rubber_loop(Win,Ptin,Ptsofar,AllPts,0):-           % while no key is pressed
    !,
    wait_click(Win,Y,X,Mod),                       % mouse click signals new line
    rubber_band(Win,Ptin,Ptout),
    rubber_loop(Win,Ptout,[Ptout|Ptsofar],AllPts,Mod).

rubber_loop(Win,LastPt,AllPts,AllPts,Mod).         % finish if non-zero Mod
```

`wait_click` waits for the user to press the mouse down again inside the viewing pane, to start another line (for a description of `wait_click`, see the section on the mouse and events below). A 'modified' click is used to terminate the `rubber_loop`. This is when the user clicks the mouse with, say, the *Option* key down. In this case the `Mod` argument will be non-zero, and the second clause for `rubber_loop` will be used.

28.8 **drag_pics** - drag a list of pictures

drag_pics(*window, names, start, amount*)

ARGUMENTS

window	: atom, name of graphics window
names	: list of picture names to be dragged
start	: point pair of the form (*Y, X*), initial mouse click
amount	: variable, will be bound to a point pair giving the distance of drag

USE

To allow the user to drag a given list of pictures around the screen.

While the mouse button is down and within the viewing pane, a grey outline of the pictures will follow the mouse. This will disappear when the mouse is released.

Example:

`drag_pics` can be used to define the click mode of a dragger tool which allows the user to drag the currently selected pictures.

```
dragger(Window,Y,X,512):-                       % if Shift key is down
    !,
    select(Window,Y,X,512).                     % just toggle selection

dragger(Window,Y,X,Mod):-
    get_sel_pics(Window,Pics),                  % get currently selected pics
    drag_pics(Window,Pics,(Y,X),Delta),         % allow user to drag them
    shift_pics(Window,Pics,Delta).              % and record translations
```

Mouse Clicks and Events.

The Macintosh is an 'event-driven' system. The user drives the system by using the mouse and generating events that are recorded and interpreted by the underlying system. There are many types of events, the most important being:

click (i.e. mouse down)
release (i.e. mouse up)

As events occur the Macintosh puts them in an event queue. The following MacPROLOG primitives allow programmers access to this event queue to find out whether or not certain events have or have not occurred. Most of these primitives need the name of a graphics window to be passed as the first argument. This must be the selected window at the time of the call.

28.9 `wait_click` - wait for mouse click in viewing pane

`wait_click(`*`window, down, across, mod`*`)`

ARGUMENTS

window	: atom, name of graphics window
down	: variable, will be bound to vertical coordinate
across	: variable, will be bound to horizontal coordinate
mod	: variable, will be bound to integer, modifier value

USE

`wait_click` suspends a MacPROLOG evaluation until the user presses the mouse button down in the viewing pane of the window. It then returns the position of the mouse within the drawing area.
mod will have an integer value indicating the keyboard status at the time of the event. i.e. whether or not any of the 'modifier' keys such as *Shift* or *Option* were down when the mouse click occurred. The *mod* value can be:

0	No key down
256	Command key down
512	Shift key down
1024	Caps Lock key down
2048	Option key down

If the user presses the mouse button down outside the viewing pane, then a beep is sounded.

If *window* is not the front window at the time of the call, `wait_click` fails.

28.10 `clicked` - test if there has been a mouse click in viewing pane

`clicked(`*`window, down, across, mod`*`)`

ARGUMENTS

window	: atom, name of graphic window
down	: variable, will be bound to vertical coordinate
across	: variable, will be bound to horizontal coordinate
mod	: variable, will be bound to modifier value

USE

To test to see if a click has taken place in the viewing pane of the named graphic window. The call fails if the user has not pressed the mouse button or if *window* is not the front window. A subsequent call to `clicked` will succeed only if the user has clicked again: the 'click' is removed from the event queue by this primitive.

The `clicked` primitive enables tool programs to be interrupted by clicks in the viewing pane. For example, in a graphical trace program you might test for a click at the beginning of every new trace. If the user has clicked the mouse, a dialogue may be generated asking the user if they wish to stop the trace.

28.11 `get_mouse` - get the current position of mouse in the drawing area

`get_mouse(`*`window, down, across`*`)`

ARGUMENTS

window	: atom, name of graphicswindow
down	: variable, will be bound to vertical coordinate
across	: variable, will be bound to horizontal coordinate

USE

To find the current position of the mouse in the drawing area of the named window. If the mouse is not in the drawing area of the window, or if *window* is not the front window, the call fails.

28.12 `mouse_down` - test if mouse is still down

`mouse_down`

USE

To test if the mouse button is still pressed. If the mouse button is currently down, the call succeeds, and fails otherwise.

28.13 mouse_up - test if mouse has been released

```
mouse_up
mouse_up(window, down, across)
```

ARGUMENTS

window	: atom, name of graphics window
down	: variable, will be bound to vertical coordinate
across	: variable, will be bound to horizontal coordinate

USE

To see if the mouse has been released. If a 'mouse-up' event has occurred then the call succeeds. In the three argument form, the call returns in the `down` and `across` arguments the position of where the mouse was released in the viewing pane of the graphic window.

EXAMPLE 1

Using these primitives we outline a simple 'picture dragger' tool which finds the top picture under the mouse, and then draws a grey image of it in `xor` mode on the screen, following the mouse's movements.
By using `xor` mode, we can remove the previous image of the picture by redrawing it in its old position before we draw its new image.
We use fast drawing (`draw_pic` rather than `add_pic`) to immediately change the screen's image.
In `redraw_pic` we have used `draw_pic` on a translated description of the picture using the `trans` picture descriptor. Alternatively, we could have left the picture description alone, and offset its local origin.

```
dragger(Window,Y,X,Mod) :-
    mouse_down,
    find_pic(Window, (Y,X),Name,Desc,Org),          % get picture to drag
    follow_mouse(Window,Y,X,penmode(xor,greypen(Desc)),Org, (Y,X)),
                                                     % set pen to xor and grey
    get_mouse(Window,Y9,X9),                         % get final mouse position
    gap(Y9,X9,Y,X,DY,DX),                            % calculate amount dragged
    shift_pic(Window,Name, (DY,DX)).                 % record drag


follow_mouse(Window,Y,X,Desc,Org,Init) :-
    mouse_up,
    !.                                               % if released, then terminate

follow_mouse(Window,Y,X,Desc,Org,Init) :-
    get_mouse(Window,Y9,X9),                         % get new mouse position
    redraw_pic(Window,Y,X,Y9,X9,Desc,Org,Init),      % draw new picture position
    follow_mouse(Window,Y9,X9,Desc,Org,Init).        % loop round
```

```
redraw_pic(Window,Y,X,Y,X,Desc,Org,Init):-          % picture hasn't moved
   !.
redraw_pic(Window,Y1,X1,Y9,X9,Desc,Org,(Y0,X0)):-
   gap(Y1,X1,Y0,X0,DY,DX),                          % get old drag gap
   gap(Y9,X9,Y0,X0,EY,EX),                          % get new drag gap
   draw_pic(Window,trans(DY,DX,Desc),Org),          % redraw at old posn. (to erase)
   draw_pic(Window,trans(EY,EX,Desc),Org).          % draw at new position

gap(Y0,X0,Y9,X9,DY,DX):-
   DY is Y0--Y9,                                    % use integer subtraction
   DX is X0--X9.                                    % use integer subtraction
```

EXAMPLE 2

We now outline the basis of a simple marqui tool.

```
my_marqui(Window,Y,X,box(T,L,D,W)):-
   mouse_down,
   marqui_loop(Window,Y,X,Y,X),
   get_mouse(Window,Y9,X9),                         % get final position of mouse
   reorder(Y,Y9,T,D),                               % reorder coordinates
   reorder(X,X9,L,W).


marqui_loop(Window,Y,X,Y1,X1):-                     % if released then terminate
   mouse_up,
   !.

marqui_loop(Window,Y,X,Y1,X1):-                     % while the mouse is down
   get_mouse(Window,Y9,X9),                         % get new position
   redraw_pic(Window,Y,X,Y1,X1,Y9,X9),              % update screen image
   marqui_loop(Window,Y,X,Y9,X9).                   % loop round


redraw_pic(Window,Y,X,Y0,X0,Y0,X0):-                % mouse didn't move
   !.

redraw_pic(Window,Y,X,Y1,X1,Y9,X9):-
   draw_box(Window,Y,X,Y1,X1),                      % draw old box
   draw_box(Window,Y,X,Y9,X9).                      % draw new box


draw_box(Window,Y0,X0,Y9,X9):-                      % draw in xor mode, grey box
   reorder(Y0,Y9,T,D),
   reorder(X0,X9,L,W),
   draw_pic(Window,greypen(penmode(xor,box(T,L,D,W)))).


reorder(A,B,A,Gap):-                                % maybe upwards drag
   A < B,
   !,
   Gap is B--A.                                     % use integer subtraction

reorder(A,B,B,Gap):-
   Gap is A--B.                                     % use integer subtraction
```

Note: mouse_pos will fail if the mouse goes outside the viewing pane, causing dragger and my_marqui to fail. This will leave a copy of the last grey box on the screen, which will go away on the next refresh. Thus, on failure, we should invalidate the viewing pane. In practice, this failure may be undesirable, and we could introduce an extra program, mouse_in_view, as defined below, to replace mouse_pos, so that if the mouse is dragged outside the viewing pane, we will get a series of beeps until it is brought back into the viewing pane.

```
mouse_in_view(Window,Y,X) :-                % get next mouse position
    get_mouse(Window,Y,X),                  % mouse is in viewing pane
    !.

mouse_in_view(Window,Y,X) :-                % first clause failed
    mouse_in_view(Window,Y,X).              % just loop and try again !
```

This idea can be extended to restricting the 'acceptable' mouse coordinates to those inside any given Rectangle in the drawing area or viewing pane.

For example:

```
restrict_mouse(Window,Y,X,Rectangle) :-     % get a valid mouse position
    get_mouse(Window,Y,X),                  % get next mouse position
    pt_in_box(Y,X,Rectangle),               % mouse inside rectangle ?
    !.

restrict_mouse(Window,Y,X,R) :-             % first clause failed
    restrict_mouse(Window,Y,X,R).           % just loop and try again !
```

The Rectangle term is of the form box(*T*,*L*,*D*,*W*)
pt_in_box is a primitive described later in this chapter, which tests to see if a point lies inside a given rectangle.

Edit Tools

The following section is for programmers wishing to write tools that allow keyboard input of text into graphic windows using edit fields.

Associated with each graphic window is a 'movable' text editor. This editor can be invoked at any position within the viewing pane. However, at any one time at most *one* edit field can be active, and all keystrokes will be entered into this edit field. If and when this editor is closed and moved to another place any text in the edit field must be extracted and saved. You will of course usually arrange for this text to remain displayed on the screen in the position it was originally typed

There are two primitives that enable you to develop tools that open, prefill and extract information from edit fields in graphics windows:

`edit_line`	for a line of text
`edit_box`	for several lines of text in a rectangular box.

The edit field is closed by a mouse click outside the edit field but still in the graphic window. A click outside the graphic window will not close the edit field.

If the user clicks in either of the scroll bars or the viewer, then the viewing pane will scroll as usual. The edit field is still active (even though its screen position may have changed).

Clicks elsewhere will be dealt with by the system, for example the selection of pull-down menu items. In the particular case of the **Fonts** menu, the edit field's text will be updated to reflect any change in font, style or size.

If the click generates a dialogue or selects another window, then the whole graphic window is deactivated, but the edit field will still be active when the graphic window is reselected,. You can override this behaviour by including a call to a `shut_down_edit` program in your `actdeact` program. This program would see which tool is currently selected, and if it was an 'edit' tool, call its `close_edit` program directly.

If a tool has opened an edit field in a graphic window then the tool's `close_edit` mode program is called by the system when the edit field is closed. This call should explicitly extract the entered text using the `get_text` primitive, otherwise it will be lost. Of course, it is the responsibility of the programmer to store the text appropriately.

Note
Due to the limitation on the length of an atom, the maximum amount of text that can be edited in any single edit field is 255 characters.

28.14 **edit_box** - open an edit rectangle

`edit_box(window, rectangle, just)`
`edit_box(window, rectangle, just, text)`

ARGUMENTS

window	: atom, name of graphics window
rectangle	: rectangle of the form box(*T, L, D, W*)
just	: integer, text justification
text	: atom, initial text for edit field

USE

Opens an edit field in the graphics window with the position and size indicated by the *rectangle* argument. Any text typed at the keyboard will now automatically appear in this box.

The text font, face and size to be used can be set by the `set_gfont` primitive described below, otherwise the window's current font, face and size will be used (which may be changed by the user from the **Fonts** menu).

The *just* argument must be one of the integers 0, 1 or -1, representing left, centre or right justified text respectively.

The four argument form of `edit_box` initialises the edit field with *text*. This allows an 'old' edit field to be reactivated, for example. If the *text* argument is not given, the edit field will be displayed initially empty.

EXAMPLE 1

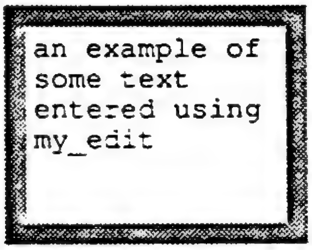
The following program defines a tool `my_edit` which opens an edit field. When the edit field is closed, the tool takes the contents of the edit field and adds a picture of it as a user-defined picture, which in this case is the text inside a grey box, 'inset' to resemble a display border.

```
my_edit(Window,Y,X,Mod) :-
   marqui(Window,(Y,X),Rect),              % get user's marqui
   edit_box(Window,Rect,0).                % open new edit rectangle

my_edit(close_edit,Window) :-              % close edit field
   get_text(Window,Details),               % get details of edit field
   ument: E 28 Tool Buildin                % create a picture description
   gensym(my_edit_box,Name),               % generate a name for new pic
   add_pic(Window,Name,Desc).              % and add it to the window


make_desc(text(Font,Face,Size,EdBox,Text),
   user_pic(Font,Size,Face,EdBox,Text,Border)) :-
                                           % return the picture to add
   inset_box(EdBox,(-10,-10),Border).      % get size of outer border

user_pic(Font,Size,Face,box(T,L,D,W),Text,Border),
[textbox(Font,Size,Face,T,L,D,W,0,Text),greypen(thick(Border))]).
```

Notes

1. `user_pic` is the definition of the user defined form. It takes 6 arguments and returns an aggregate description of some text and a box. `Border` is always a term of the form `box(T,L,D,W)`.

2. The `get_text` primitive (described later) returns `Details` from the edit field on closing.



EXAMPLE 2

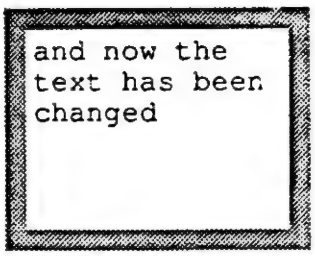
To re-edit a text picture which was added using `my_edit`, we could add a new first clause for `my_edit` as follows.

```
my_edit(Window,Y,X,Mod):-                    % re-edit 'my_edit' picture
    find_pic(Window,Y,X,Name,Desc,Org),      % get picture under mouse
    Desc=[user_pic(Font,Size,Face,EdBox,Text,Border)],% is it one of mine ??
    !,
    del_pic(Window,Name),                    % if so, delete it
    set_gfont(Window,Font,Face,Size),        % set up the edit font
    edit_box(Window,EdBox,0,Text).           % open edit rectangle with initial Text
```

Notes

1. We use `set_gfont` to set up the edit field font details, see below.
2. In the above program we look at the description of the picture under the mouse click, and if we recognise it as one of ours, we remove it and open an edit field initialised with the text found in its description. Then we rely on the previously given `close_edit` program to add a new picture with the edited text when the edit field is closed.

These programs as stated do not preserve the original names for the edited pictures nor their position in the window. We could preserve all previous information by setting some properties (using `set_prop`) when opening the edit field to indicate what state we are in, and then extending the `close_edit` program.



28.15 `edit_line` - set up editable line of text

```
edit_line(window,down,across,just)
edit_line(window,down,across,just,text)
```

ARGUMENTS

window	: atom, name of graphics window
down	: integer, horizontal coordinate
across	: integer, vertical coordinate
just	: integer, justification
text	: atom, initial text for edit field

USE

1. Four argument use To open an initially empty edit line of text into which the user may type. The width of the edit field will expand as characters are entered.

2. Five argument use To set up an editable line of text, initialised with `text`, into which any subsequent keyboard input will go. The width of the edit field will initially be enough to display the `text`, and will expand as new characters are entered. It uses the current font, face and size. This form of `edit_line` is generally used to re-edit a line of text in a similar way to the four argument form of `edit_box` (see above).

EXAMPLE

A tool that creates a single line text picture of some entered text might be defined as follows.

```
new_edit(Window,Y,X,Mod) :-
    edit_line(Window,Y,X,0).                      % open an edit line

new_edit(close_edit,Window) :-                    % close the edit line
    get_text(Window,Details),                     % extract the text details from the edit
    make_desc2(Details,Picture),                  % construct a description
    gensym(my_edit_item,Name),                    % generate a name for new picture
    add_pic(Window,Name,Picture).                 % add the picture to the window

make_desc2(text(Font,Face,Size,box(T,L,D,W),Text),
      text(Font,Size,Face,T,L,Text)).             % return the description of a line of text
```

(`get_text` returns the `Details` from the edit field on closing - see below.)

28.16 `get_text` - return text editor details

`get_text(`*`window, details`*`)`

ARGUMENTS

window	: atom, name of graphic window
details	: variable, will be unified with a term giving text details

USE

To extract the *details* of the text entered in an edit field. This will be used in the `close_edit` program for an edit tool to enable a picture to be constructed based on the text entered in the edit field. *details* will be unified with a term of the form

```
text(Font,Face,Size,box(T,L,D,W),Atom)
```

where the `box` term describes the enclosing rectangle of the text and *Atom* is the text.

Note
Due to the limitations on the length of an atom, the maximum amount of text that can be extracted is 255 characters. If more text than this has been entered, only the first 255 characters will be returned in *Atom*. This effectively sets a limit on the size of an edit field.

28.17 `get_gfont` - get current font details for graphic window

`get_gfont(`*`window, font, face, size`*`)`

ARGUMENTS

window	: atom, name of graphics window
font	: variable, will be bound to name of font
face	: variable, will be bound to font face
size	: variable, will be bound to point size of font

USE

To get the current font details of a graphic window. This is useful for example, to implement user defined pictures which reflect the current font.

28.18 **`set_gfont`** - set font details for edit field

`set_gfont(`*`window, font, face, size`*`)`

ARGUMENTS

`window`	: atom, name of graphics window
`font`	: atom, name of font
`face`	: integer, font face
`size`	: integer, size of font

USE

To set the font for a graphic window and, if open, its current edit field.

The Anatomy of a Font

Each character in a font is defined by pixels arranged in rows and columns. This pixel arrangement is called a character image.

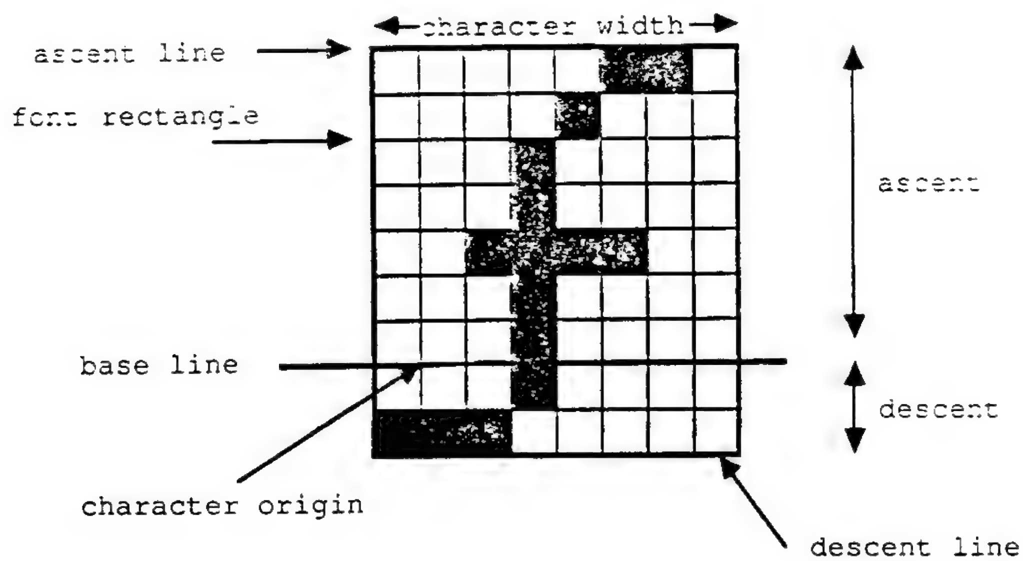
The base line is a horizontal line coincident with the bottom of each character, excluding descenders (the 'tails' of letters such as y, g or j). The character origin is a point on the base line used as a reference location for drawing the character.

Conceptually the base line is the line that the pen is on when it starts to draw a character, and the character origin is the point where the pen starts drawing.

The *`ascent`* is the distance from the base line to the top of the font rectangle, and the *`descent`* is the distance from the base line to the bottom of the font rectangle.

`maxwidth` is the maximum character width for the font, and is therefore the greatest distance the pen will move when a character is drawn.

The *`leading`* is the amount of blank space to draw between lines of single-spaced text -- the number of pixels between the descent line of one line of text and the ascent line of the next line of text.



28.19 `font_info` - get characteristics of current font

`font_info(ascent,descent,maxwidth,leading)`

ARGUMENTS

ascent	: variable, will be bound to ascent height of current font
descent	: variable, will be bound to descent height of current font
maxwidth	: variable, will be bound to maximum width of current font
leading	: variable, will be bound to distance between lines

USE

To find out the characteristics of the current font.

28.20 `text_width` - get width of atom for a given font

`text_width(text,font,face,size,width)`

ARGUMENTS

text	: atom, text string
font	: atom, name of font
face	: integer, font face
size	: integer, size of font
width	: variable, will be bound to pixel width of atom

USE

To find out the pixel *width* of some `text` using a given *font, face* and *size*. This is useful in calculating how much space to allow for text which is to be inserted in a text based picture. The *width* is the sum of all the individual character widths.

Miscellaneous Rectangle Primitives

MacPROLOG provides a set of miscellaneous primitives to manipulate rectangle terms efficiently.

28.21 **`inset_box`** - inset a rectangle

`inset_box(`*`rect1,point,rect2`*`)`

ARGUMENTS

rect1	: a rectangle of the form box (*T, L, D, W*)
point	: a point pair of the form (*DY, DX*)
rect2	: variable, will be bound to a rectangle term

USE

To 'inset' a rectangle in a positive or negative direction. This shrinks or expands the initial rectangle. The left and right sides of the rectangle *rect1* are moved in towards the centre of the rectangle by *DX*; the top and bottom are moved towards the centre by *DY*. If *DY* or *DX* are negative, the appropriate pair of sides is moved outwards instead of inwards. The effect is to alter the depth and width of the *rect1* by 2**DY* vertically and 2**DX* horizontally, with the resulting rectangle, *rect2*, remaining centred in the same place as *rect1*.

Examples

The call

```
inset_box(box(0,0,100,100),(20,30),final_box)
```

would result in `final_box` being bound to the term

```
box(20,30,60,40).
```

The call

```
inset_box(box(15,15,100,100),(-20,-30),final_box)
```

would result in `final_box` being bound to the term

```
box(-5,-15,125,145).
```

28.22 **`offset_box`** - offset a rectangle

`offset_box(`*`rect1,point,rect2`*`)`

ARGUMENTS

rect1	: a rectangle of the form box (*T, L, D, W*)
point	: a point pair of the form (*DY, DX*)
rect2	: variable, will be bound to a rectangle

USE

To 'offset' a rectangle in a positive or negative direction. This moves the initial rectangle in both the vertical and horizontal directions. *rect1* is moved by *DY* vertically and by *DX* horizontally. If *DY* or *DX* are positive, then the movement is to the right and down; if either is negative, the corresponding movement is in the opposite direction. The rectangle retains its shape and size; only its position changes.

Examples

The call

```
offset_box(box(0,0,100,100),(20,30),final_box)
```

would result in `final_box` being bound to the term

```
box(20,30,100,100).
```

The call

```
offset_box(box(0,0,100,100),(-20,-30),final_box)
```

would result in `final_box` being bound to the term

```
box(-20,-30,100,100).
```

28.23 `intersect_box` - form the intersection of two rectangles

`intersect_box(rect1,rect2,rect3)`

ARGUMENTS

rect1	: a rectangle of the form `box(T,L,D,W)`
rect2	: a rectangle
rect3	: variable, will be bound to a rectangle

USE

To calculate the rectangle that is the intersection of two rectangles. Rectangles that touch at a line or a point are not considered intersecting, because their intersecting rectangle does not enclose any points. If the rectangles do not intersect the call fails.

Examples

The call

```
intersect_box(box(0,0,100,100),box(-50,50,100,100),final_box)
```

would result in `final_box` being set to

```
box(0,50,50,50).
```

The call

```
intersect_box(box(0,50,150,50),box(-50,50,100,150),final_box)
```

would also result in `final_box` being set to

```
box(0,50,50,50).
```

28.24 **`union_box`** - form the union of two rectangles

`union_box(`*`rect1, rect2, rect3`*`)`

ARGUMENTS

`rect1`	: a rectangle of the form `box(`*`T, L, D, W`*`)`
`rect2`	: a rectangle
`rect3`	: variable, will be bound to a rectangle

USE

To calculate the smallest rectangle that encloses two rectangles.

<u>Examples</u>

The call

```
union_box(box(0,0,100,100),box(-50,50,100,100),final_box)
```

would result in `final_box` being bound to

```
box(-50,0,150,150).
```

The call

```
union_box(box(0,50,150,50),box(-50,50,100,150),final_box)
```

would result in `final_box` being set to

```
box(-50,50,200,150).
```

28.25 pt_in_box - test if a point is in a rectangle

pt_in_box(*point, rect*)

ARGUMENTS

point	: a point pair of the form (*Y, X*)
rect	: a rectangle of the form box(*T, L, D, W*)

USE

To determine whether the pixel below and to the right of the given coordinate *point* is enclosed in the specified rectangle, *rect*. The call succeeds if the pixel lies in the rectangle, and fails otherwise. For example

```
        pt_in_box((20,30),box(0,0,40,50))              succeeds,
but
        pt_in_box((20,30),box(-10,-10,20,30))          fails.
```

Note
Because of the way that points and rectangles are handled in MacPROLOG, points on the left and top edges of a rectangle are considered to be 'inside', whilst those on the bottom and right edges are considered to be 'outside'.
For example

```
        pt_in_box((0,5),box(0,0,10,10))                succeeds,
and     pt_in_box((5,0),box(0,0,10,10))                succeeds,
but
        pt_in_box((5,10),box(0,0,10,10))               fails,
and     pt_in_box((10,5),box(0,0,10,10))               fails.
```

28.26 box_in_box - test if one rectangle is in another rectangle

box_in_box(*rect1, rect2*)

ARGUMENTS

rect1	: a rectangle of the form box(*T, L, D, W*)
rect2	: a rectangle of the form box(*T, L, D, W*)

USE

To determine whether the rectangle specified by *rect1* is enclosed in the rectangle, *rect2*. The call succeeds if the first rectangle lies in the second rectangle, and fails otherwise.
For example

```
        box_in_box(box(10,10,20,30),box(0,0,40,50))  succeeds,
but
        box_in_box(box(10,10,20,30),box(-10,-10,20,30))  fails.
```

Note that the comments made above about points on the edges of a rectangle also apply here.

29 Implementing an Application

In this chapter we explain various facilities of MacPROLOG that are useful for applications.

29.1 Execution on loading

If a program file contains a definition for a unary relation '<LOAD>' this program will be called immediately after the file is loaded. The single argument to '<LOAD>' has no meaning and will be ignored. On completion of the evaluation of the '<LOAD>' call, the '<LOAD>' program code will be deleted so that a subsequent load will not result in a second call to '<LOAD>'. The next program file you load can contain another definition for '<LOAD>' and this will be called when the file load is complete. Again the definition of '<LOAD>' will be deleted on return from the call.

This automatic calling of the '<LOAD>' program occurs after the loading of both source and object code programs.

(See also section 29.1.5 for how to save a '<LOAD>' program.)

29.1.1 Setting up new menus

The execute on load facility can be used to set up menus suitable for an application on loading of the application. For example, suppose that you had an application which needed to have two menus labelled **Database** and **Interrogate**. Include in the source of your application a definition of the form:

```
'<LOAD>'(Ignored_arg) :-
 install_menu('Database', ..... /* list of menu items */),
 install_menu('Interrogate',..... /* list of menu items */).
```

install_menu is a primitive for manipulating the menu bar, described in the chapter on Menu Handling. Remember that '<LOAD>' *must* be defined as a unary relation even though the argument has no role.

On loading this application, menus named **Database** and **Interrogate** will be added to the menu bar. As explained in the chapter on Menu Handling, your application should also contain definitions for the relations 'Database' and 'Interrogate' because these relations will be called when items from these menus are selected.

For example, if the **Database** menu contains an item **Enter**, which is to allow users to enter data via a modal dialogue, your program should contain a definition of the form:

```
'Database'('Enter'):-
  mdialog(....),          /*call to dialogue construction primitive
                          with parameters describing the dialogue */
  add_data(....).         /* call to remember the entered data */
```

29.1.2 Changing menus

As well as extending the menu bar, you can also delete or modify existing menus on loading. Just include the appropriate calls to the menu handling primitives in your '<LOAD>' definition.
For example, to remove **Find definition...** from the **Find** menu, include the call

```
install_menu('Find',['What to find.../W',
                     'Replace & find next/R',
                     'Find next/F',
                     'Find selection/E'])
```

The use of exactly the same names for the preserved menu items is important. In order to completely remove the **Find** menu without giving alternatives give the call:

```
kill_menu('Find')
```

WARNING: do not delete or change the **File** or menus. These menus must remain as they are. They provide the interface to the Mac system and the entries and their relative order must be preserved. You can delete items from or completely remove any of the other MacPROLOG menus.

29.1.3 Extending the MacPROLOG menus

You can also extend menus on loading by including calls to `extend_menu`. If you use this facility to extend a MacPROLOG menu you must implement the new menu item. Normally you would do this by providing a new definition of the menu title relation in the loaded program.

However, you cannot modify the definitions of the `'Edit'`, `'Find'`, `'Windows'` and `'Eval'` relations as their definitions are protected. For an extension of any of these menus, say with an item **New command**, just include a definition for the no argument relation `'New command'` in your program. When the **New command** item is selected, your definition for it will be called. This is because each of the **Edit**, **Find**, **Windows** and **Eval** menu definitions has a last clause to handle unrecognised menu items by calling the item.

For example, `'Eval'` has a last clause

```
'Eval'(Item):-Item.
```

29.1.4 Defining operators

An application that needs to use other than the predefined operators for user input or for message output must use a '<LOAD>' program to declare the operators on loading. Just include the appropriate sequence of op calls in the body of the '<LOAD>' clause.

For example, the definition

```
'<LOAD>'(X) :-
    op(600,xfx,@),
    op(550,fx,[$,#]).
```

will set up `@` as a non-associative infix operator, and `$` and `#` as non-associative prefix operators.

29.1.5 Saving the object code of a '<LOAD>' definition

As described above, when you load a program with a source definition of `'<LOAD>'` this definition will be executed on completion of the load. However, because the object code of the definition of the relation `'<LOAD>'` is deleted immediately after it has been executed, there will be no definition of `'<LOAD>'` if you save the object code immediately after loading the source.

Before you save the program in object code form, you must force a recompilation of the program window in which `'<LOAD>'` is defined. You can do this by adding and deleting a space in the window and then recompiling using the **Check program** command of the **Eval** menu. Now an object code definition of `'<LOAD>'` will be in the memory of the computer ready for saving as part of a compiled application.

29.2 Modifying the commands of the File menu

Although you cannot alter or remove the **File** menu you can change the effect of the **Reinitialise...** and **Quit** commands of the menu.

If you have a definition for a 0-argument relation `'<REINIT>'` this program will be called when the **Reinitialise...** command is selected. The normal dialogue allowing you to delete all source windows or all source windows and all currently loaded object code will not be displayed.

Similarly, you can have a definition for a *unary* relation `'<QUIT>'`. As with `'<LOAD>'` the single argument is ignored. This program will be called when the **Quit** command is selected and before MacPROLOG is exited. With a suitable definition for `'<QUIT>'`, an application can save the state of the clause data base or the contents of certain windows before the exit.
Even if you provide a definition for `'<QUIT>'` you will still be asked if you want to save your program should you have typed into any of its edit windows since the last **Save...** If you elect to do the save, your `'<QUIT>'` program will be called *after* the save is completed.

NOTE : You cannot prevent the exit from MacPROLOG once the **Quit** comand has been selected. You will exit even if your `'<QUIT>'` program fails or aborts.

29.3 Error handling

When an evaluation error occurs a MacPROLOG program is called to handle the error. The program is called with two arguments, an integer error number and the call term. If you include in your application a definition of the relation `'<ERROR>'`, which takes the same two arguments, your program will be called instead of the default error handler.

As an example, the following program defines a user error handler very similar to the one supplied in the MacPROLOG programming environment (see also the section below on the `errmess` primitive).

```
'<ERROR>'(Id, Call):-
    errmess(Id,Message), !,
    mdialog(50, 50,200,380,
    [button( 5,305, 20, 65,'Succeed'),
     button(35,305, 20, 65,'Fail'),
     button(65,305, 20, 65,'Stop'),
       text( 5, 60, 34,240,wseq(['ERROR:',Message]),
       text(41, 60, 52,240,writeq(Call)),
      icon( 5,  5, 80, 52, 0),
    Btn),
    Btn = 2 -> abort.
```

The `!` immediately after the call to `errmess` is necessary to avoid picking up the default error message on backtracking.

29.3.1 The `errmess` primitive

`errmess` is a primitive that you can use to pick up an appropriate message string for the identified error. See Appendix A for the `errmess` program.

Notice in particular that 2 is the identifier for the error 'No definition for relation'. If you want this error to automatically result in a failure of the call without the display of an error message dialogue, add a first clause:

```
'<ERROR>'(2,Call):-!,fail.
```

to your error handler program. You might also want to handle error 13 in this way.

29.4 Shipping an application

You *must* obtain a special licence from LPA to enable you to distribute an **LPA MacPROLOG™** application. If you would like further details of this licence please write to:

OEM Licences
Logic Programming Associates Ltd.
Studio 4
The Royal Victoria Patriotic Building
Trinity Road
London SW18 3SX
England.

With the licence you will be supplied with a special runtime version of MacPROLOG that you can include with your distributed application. You will also receive instructions on how to modify your application so that it will run outside the MacPROLOG programming environment, using only the runtime version of MacPROLOG.

Appendices

Appendix A
The **errmess** Primitive

```
errmess(0,'Number argument needed').
errmess(1,'Arithmetic error').
errmess(2,'No definition for relation').
errmess(3,'Incomplete argument list').
errmess(4,'Invalid predicate symbol').
errmess(5,'Invalid form of call').
errmess(6,'Syntax error').
errmess(7,'Error in compilation').
errmess(8,'Structure too deep').
errmess(9,'Error in assembly').
errmess(10,'Error in lambda').
errmess(11,'Bad lambda expression').
errmess(12,'Format expression error').
errmess(13,'Invalid form of use').
errmess(14,'Cannot add clause for compiled relation').
errmess(15,'New window name already used').
errmess(16,'Cannot compile program for interpreted relation').
errmess(17,'Cannot compile program for primitive').
errmess(18,'Error in graphics').
errmess(19,'Error in printing').
errmess(20,'Error in decompilation').
errmess(21,'Text overflow in window (max 32K)').
errmess(22,'Invalid picture description').
errmess(23,'Duplicate picture name').
errmess(24,'No definition for graphics verb').
errmess(-33,'Directory is full').
errmess(-34,'Disk is full').
errmess(-35,'No such volume').
errmess(-36,'Disk I/O error').
errmess(-37,'Bad name error ¿¿¿').
errmess(-38,'File not open').
errmess(-39,'Read past end of file').
errmess(-40,'Invalid file position').
errmess(-42,'Too many files').
errmess(-43,'File not found').
errmess(-44,'Disk write protected').
errmess(-45,'File is locked').
errmess(-46,'Volume is locked').
errmess(-47,'File is already open').
errmess(-48,'Renamed file already exists').
errmess(-49,'File already open for writing').
errmess(-51,'Bad path id').
errmess(-52,'Seek error').
errmess(-53,'Volume not on line(ejected)').
errmess(-54,'File permission error').
errmess(-56,'No such drive').
errmess(-59,'SERIOUS ERROR - PLEASE REPORT').
errmess(-60,'Directory corrupted').
errmess(-61,'Write permission error').

errmess(Id,Message):-
    concat('error: ',Id,Message).
```

Appendix B
ASCII Character codes on the Macintosh

<table>
<tr><td>space - 32</td><td>A - 65</td><td>a - 97</td><td>Ä - 128</td><td>† - 160</td><td>¿ - 192</td></tr>
<tr><td>! - 33</td><td>B - 66</td><td>b - 98</td><td>Å - 129</td><td>° - 161</td><td>¡ - 193</td></tr>
<tr><td>" - 34</td><td>C - 67</td><td>c - 99</td><td>Ç - 130</td><td>¢ - 162</td><td>¬ - 194</td></tr>
<tr><td># - 35</td><td>D - 68</td><td>d - 100</td><td>É - 131</td><td>£ - 163</td><td>√ - 195</td></tr>
<tr><td>$ - 36</td><td>E - 69</td><td>e - 101</td><td>Ñ - 132</td><td>§ - 164</td><td>ƒ - 196</td></tr>
<tr><td>% - 37</td><td>F - 70</td><td>f - 102</td><td>Ö - 133</td><td>• - 165</td><td>≈ - 197</td></tr>
<tr><td>& - 38</td><td>G - 71</td><td>g - 103</td><td>Ü - 134</td><td>¶ - 166</td><td>∆ - 198</td></tr>
<tr><td>' - 39</td><td>H - 72</td><td>h - 104</td><td>á - 135</td><td>ß - 167</td><td>» - 199</td></tr>
<tr><td>(- 40</td><td>I - 73</td><td>i - 105</td><td>à - 136</td><td>® - 168</td><td>« - 200</td></tr>
<tr><td>) - 41</td><td>J - 74</td><td>j - 106</td><td>â - 137</td><td>© - 169</td><td>… - 201</td></tr>
<tr><td>* - 42</td><td>K - 75</td><td>k - 107</td><td>ä - 138</td><td>™ - 170</td><td></td></tr>
<tr><td>+ - 43</td><td>L - 76</td><td>l - 108</td><td>ã - 139</td><td>´ - 171</td><td></td></tr>
<tr><td>, - 44</td><td>M - 77</td><td>m - 109</td><td>å - 140</td><td>¨ - 172</td><td></td></tr>
<tr><td>- - 45</td><td>N - 78</td><td>n - 110</td><td>ç - 141</td><td>≠ - 173</td><td></td></tr>
<tr><td>. - 46</td><td>O - 79</td><td>o - 111</td><td>é - 142</td><td>Æ - 174</td><td></td></tr>
<tr><td>/ - 47</td><td>P - 80</td><td>p - 112</td><td>è - 143</td><td>Ø - 175</td><td></td></tr>
<tr><td>0 - 48</td><td>Q - 81</td><td>q - 113</td><td>ê - 144</td><td>∞ - 176</td><td></td></tr>
<tr><td>1 - 49</td><td>R - 82</td><td>r - 114</td><td>ë - 145</td><td>± - 177</td><td></td></tr>
<tr><td>2 - 50</td><td>S - 83</td><td>s - 115</td><td>í - 146</td><td>≤ - 178</td><td></td></tr>
<tr><td>3 - 51</td><td>T - 84</td><td>t - 116</td><td>ì - 147</td><td>≥ - 179</td><td></td></tr>
<tr><td>4 - 52</td><td>U - 85</td><td>u - 117</td><td>î - 148</td><td>¥ - 180</td><td></td></tr>
<tr><td>5 - 53</td><td>V - 86</td><td>v - 118</td><td>ï - 149</td><td>µ - 181</td><td></td></tr>
<tr><td>6 - 54</td><td>W - 87</td><td>w - 119</td><td>ñ - 150</td><td>∂ - 182</td><td></td></tr>
<tr><td>7 - 55</td><td>X - 88</td><td>x - 120</td><td>ó - 151</td><td>Σ - 183</td><td></td></tr>
<tr><td>8 - 56</td><td>Y - 89</td><td>y - 121</td><td>ò - 152</td><td>Π - 184</td><td></td></tr>
<tr><td>9 - 57</td><td>Z - 90</td><td>z - 122</td><td>ô - 153</td><td>π - 185</td><td></td></tr>
<tr><td>: - 58</td><td>[- 91</td><td>{ - 123</td><td>ö - 154</td><td>∫ - 186</td><td></td></tr>
<tr><td>; - 59</td><td>\ - 92</td><td>| - 124</td><td>õ - 155</td><td>ª - 187</td><td></td></tr>
<tr><td>< - 60</td><td>] - 93</td><td>} - 125</td><td>ú - 156</td><td>º - 188</td><td></td></tr>
<tr><td>= - 61</td><td>^ - 94</td><td>~ - 126</td><td>ù - 157</td><td>Ω - 189</td><td></td></tr>
<tr><td>> - 62</td><td>_ - 95</td><td>del - 127</td><td>û - 158</td><td>æ - 190</td><td></td></tr>
<tr><td>? - 63</td><td>` - 96</td><td></td><td>ü - 159</td><td>ø - 191</td><td></td></tr>
<tr><td>@ - 64</td><td></td><td></td><td></td><td></td><td></td></tr>
</table>

Appendix C
MacPROLOG Reserved Property/Object Names

You should not use any of the following as a property name. They are used by MacPROLOG for special purposes. If you use them for your own properties you may affect the behaviour of a MacPROLOG primitive, or a menu command of the programming environment.

```
'ARITY'
'BOX'
'CLAUSES'
'CLIPPIC'
'CURSOR'
'DATAREL'
'DEFINWIN'
'ECHOT'
'GRAPHIC'
'HELP'
'INFOWIN'
'INPUT'
'MODE'
'MODULE'
'OLD'
'OUTPUT'
'DISPLAY'
'DISPLAYT'
'INPUT'
'INPUTT'
'PATH'
'QNAME'
'QPIC'
'QUERY'
'SNAME'
'SPYPOINT'
'SUFFIX-VALUE'
'SYNTAX'
'WINDOW'
'?STEP-FLAG?'
'?STEP-QUERY?'
'<OP>'
```

The following are reserved Object names.

```
'DGW'
'OUTWIN'
'?CLIP-PIC?'
```

Appendix D
MacPROLOG Predeclared Operators

The following are the predeclared operators of MacPROLOG Edinburgh syntax.

Priority	Type	Name
1200	xfx	:-
1200	xfx	-->
1200	fx	:-
1200	fx	?-
1150	fx	mode
1150	fx	public
1100	xf	?
1100	xfy	;
1100	xfy	|
1050	xfy	->
1000	xfy	,
900	fy	not
900	fy	spy
900	fy	nospy
700	xfx	=
700	xfx	\=
700	xfx	is
700	xfx	=..
700	xfx	\==
700	xfx	=:=
700	xfx	=\=
700	xfx	<
700	xfx	>
700	xfx	=<
700	xfx	>=
700	xfx	≥
700	xfx	≤
500	yfx	+
500	yfx	++
500	yfx	-
500	yfx	--

Priority	Type	Name
500	yfx	/\
500	yfx	\/
500	fx	+
500	fx	-
500	fx	\
400	yfx	**
400	yfx	*
400	yfx	/
400	yfx	//
400	yfx	«
400	yfx	»
400	yfx	<<
400	yfx	>>
300	xfx	mod
200	xfy	^

Appendix E
Compatibility with Quintus PROLOG

The following primitives of Quintus PROLOG are not supported in LPA MacPROLOG™.

```
absolute_file_name
ancestors
break
'C'
character-count
compile
current_atom
current_input
current_key
current_output
current_predicate
current_stream
debug
debugging
depth
ensure_loaded
erase
expand_term


fileerrors
flush_output
foreign
foreign_file
format
gc
gcguide
incore
instance
leash
library_directory
line_count
line_position
load_foreign_files

manual
maxdepth
nodebug
nofileerrors
nogc
no_style_check
```

```
open_null_stream

portray
print
prompt
reconsult
recorda
recorded
recordz
reinitialise
restore
save_program
set_input
set_output
source_file
statistics
stream_code
stream_position
style_check
subgoal_of

term_expansion
trimcore
ttyflush
ttyget
ttyget0
ttynl
ttyput
ttyskip
ttytab

unix
user-help
vars

write_canonical
```

Appendix F
Graphic Editor Source

This 'Graphic Editor' program is included as source code on the LPA MacPROLOG disk ('Editor Source'). It demonstrates the various graphics, menu and dialogue primitives available in MacPROLOG, and it also serves as an example of a complete graphics application written in MacPROLOG.

When you run this program a help file is loaded at the beginning giving instructions on its use. You will see the 'Graphic Editor' graphic window, which is created by this program, together with its various tools which draw and manipulate shapes.

The complete source text of the 'Graphic Editor' is reproduced here with additional comments and explanations.

There are six program windows in the source text. These are

Graphic Editor : start
Graphic Editor : menus
Graphic Editor : dialogues
Graphic Editor : tools
Graphic Editor : rubber band
Graphic Editor : misc.

F.1 Graphic Editor : start

This window contains the startup code for the program. We use the execute on load facility by defining the relation `'<LOAD>'` (which takes a single dummy argument).

F.1.1

The `'<LOAD>'` program creates the **Graphic Editor** graphic window (by calling `initial_gw`) and then loads the help file using the `tload` primitive. This 'help' window is then resized, its font changed to Courier 10 point, made visible, and then finally brought to the front. (See the Window Handling chapter for details of these primitives.)

```
'<LOAD>'(Dummy):-
    initial_gw('Graphic Editor'),
    tload('Editor Help'),
    wsize('Editor Help', 70, 55, 215, 365),
    wfont('Editor Help','Courier',0,10),
    wshow('Editor Help'),
    wfront('Editor Help').
```

F.1.2

The `initial_gw` program calls `wgcreate` to create the **Graphic Editor** window. The `init_gensym` calls are to initialise the picture names for the various graphical objects which will be created. The 'global values' of the program are stored as properties, which are initialised here using `set_prop`. The window's activation / deactivation program is defined to be `menus`, and `winTools` adds the graphic tools to the window (see below).

```
initial_gw(Name):-
    wgcreate(Name, 50, 30, 250, 400, 96, 300, 300, 0, 1),
    init_gensym(userRRect),
    init_gensym(userRect),
    init_gensym(userLines),
    init_gensym(userPoly),
    init_gensym(userOval),
    set_prop('Graphic Editor', 'PENSIZE',  thin),
    set_prop('Graphic Editor', 'PENPAT', blackpen),
    set_prop('Graphic Editor', 'START', 315),
    set_prop('Graphic Editor', 'ANGLE', 90),
    set_prop('Graphic Editor', 'FILLSWITCH', filled),
    set_prop('Graphic Editor', 'FILL',  lgrey),
    set_prop('Graphic Editor', 'OVALDEPTH', 30),
    set_prop('Graphic Editor', 'OVALWIDTH', 30),
    gactdeact(Name, menus),!,
    winTools(Name),
    wfront(Name).
```

F.1.3

The 'actdeact' program for the window defines what happens when the window is activated (i.e. becomes the front window) or deactivated. Here we simply kill the Graphic Editor menus when the window is deactivated, and reinstate them when the window is activated. This means the menus only appear on the menu bar when the **Graphic Editor** window is the front window.

```
menus(activate, Name):-
          penMenu,
          fillMenu,
          roundMenu.

menus(deactivate, Name):-
          kill_menu('Round'),
          kill_menu('Pen'),
          kill_menu('Fill').
```

F.1.4

Tools are added to the window using the `add_tools` graphics primitive. A tool is defined by giving its program name and its graphical description, in a term of the form

`tool_program(graphical_description)`

Note the user defined graphical forms for the graphical descriptions of some of these tools (`lettera`, `myrect`, etc.). The `select`, `drag`, `enter_text`, `pic_info` and `eraser` tools are all predefined in MacPROLOG (see the Graphic Windows chapter). The rest are defined in the tools window of this program.

```
winTools(Window):-
     add_tools(Window,
    [select(arrow()),
     drag(dragger()),
     geLines([
           double([lines([(28,5),(10,18),(16,26),(10,18),(28,5)]),
                   fillcircle(28,5,2),
                   fillcircle(10,18,2),
                   fillcircle(16,26,2),
                 ]),
           greypen(lines([(28,5),(16,26),(28,5)])),
            ]),
     enter_text(lettera(Window)),
     pic_info(info_icon()),
     geRect(myrect(14,4,14,24)),
     geRRect(myrrect(14,4,14,24)),
     geOval(myoval(14,4,14,24)),
     geArc(myarc(14,4,14,24)),
     gePoly(double(grey(
           fillpoly([(16,2),(24,30),(2,10),(30,10),(8,30)])))),
     eraser(rubber())
    ], 3).
```

F.1.5

These are the user defined graphical forms which describe the graphical representations of the window's tools. Note that the `get_pen_details` call (defined later) returns the current settings of the fill pattern, pen size etc.

```
lettera(Name,Desc):-
     get_gfont(Name,Fname,Fc,Size),
     Desc = text(Fname,Size,Fc,28,12,'A').


myrect(T,L,D,W,Desc):-
     get_pen_details(Size,Penpattern,Fillswitch,Pattern),
     on(Fillswitch-Prim,[hollow-box, filled-fillbox]),
     Desc = Penpattern(thin(Pattern(Prim(T,L,D,W)))).


myrrect(T,L,D,W,Desc):-
     box_details(Oh,Ow),
     //(Oh,5,Soh),                  /* Integer division */
     //(Ow,5,Sow),
     get_pen_details(Size,Penpattern,Fillswitch,Pattern),
     on(Fillswitch-Prim,[hollow-box, filled-fillbox]),
     Desc = Penpattern(thin(Pattern(Prim(T,L,D,W,Soh,Sow)))).


myoval(T,L,D,W,Desc):-
     get_pen_details(Size,Penpattern,Fillswitch,Pattern),
     on(Fillswitch-Prim,[hollow-oval, filled-filloval]),
     Desc = Penpattern(thin(Pattern(Prim(T,L,D,W)))).


myarc(T,L,D,W,Desc):-
     arc_details(S,A),
     get_pen_details(Size,Penpattern,Fillswitch,Pattern),
     on(Fillswitch-Prim,[hollow-arc, filled-wedge]),
     Desc = Penpattern(thin(Pattern(Prim(T,L,D,W,S,A)))).
```

F.2 Graphic Editor : menus

These are the menu installation programs, which will be called whenever the 'Graphic Editor' window is activated. When each menu is installed the current setting is marked.

F.2.1

The **Pen** menu offers pen size and colour options, and also allows the switching on and off of the graphic window's viewer.

```
penMenu:-
    install_menu('Pen',
                [thin,nilpen,thick,
                'Pen size...','(-;blackpen',greypen,'(-;viewer']),
    get_prop('Graphic Editor', 'PENSIZE', Size),
    markpen(Size),
    get_prop('Graphic Editor','PENPAT',Pat),
    mark_item('Pen',Pat).
```

F.2.2

The **Fill** menu offers all the fill pattern options, and also allows the selection of filled or hollow objects.

```
fillMenu:-
    install_menu('Fill',
                [filled,hollow,'(-;black',grey,lgrey,dots,check,
                crosses,diag,diamonds,horiz,rdiag,speckled,
                stripesthin, stripesthick,
                waves,white,alpha,beta]),
    get_prop('Graphic Editor','FILLSWITCH',Switch),
    get_prop('Graphic Editor','FILL',Pattern),
    mark_item('Fill',Switch),
    mark_item('Fill',Pattern).
```

F.2.3

The **Round** menu allows the selection of the 'roundedness' of round rectangles, and the form of arcs.

```
roundMenu:-
    install_menu('Round',['Rectangle...','Arc...']).
```

The actual menu programs follow. These will be called whenever the user selects one of the items from one of the menus.

F.2.4

The **Pen** menu's **viewer** option allows the window's viewer to be switched on and off. The **Pen size...** option generates a dialogue allowing selection of any pen size (the `penvals` definition is in the **dialogues** window of this program). The menu also offers two pen colours and some predefined pen sizes.

```
'Pen'(viewer):-
     wfront(Name),
     marked_item('Pen',viewer),!,
     unmark_item('Pen',viewer),
     gviewer(Name,off).

'Pen'(viewer):-
     wfront(Name),!,
     mark_item('Pen',viewer),
     gviewer(Name,on).

'Pen'(viewer):-!.

'Pen'('Pen size...'):-!,
     penvals,
     wfront(Name),
     inval_tool_pane(Name).

'Pen'(Item):-
     update_menu_selection(Item,'Pen',
                        [blackpen,greypen],'PENPAT'),!.

'Pen'(Item):-
     get_prop('Graphic Editor','PENSIZE',Size),
     unmarkpen(Size),
     mark_item('Pen',Item),
     set_prop('Graphic Editor','PENSIZE',Item),
     wfront(Name),
     inval_tool_pane(Name).


markpen(pensize(A,B)):-!,
     mark_item('Pen','Pen size...').
markpen(Size):-
     mark_item('Pen',Size).


unmarkpen(pensize(A,B)):-!,
          unmark_item('Pen','Pen size...').
unmarkpen(Size):-
          unmark_item('Pen',Size).
```

F.2.5

The **Fill** menu allows the selection of either filled or hollow objects, and the selection of a fill pattern.

```
'Fill'(Item):-
    update_menu_selection(Item,'Fill',[filled,hollow],
                          'FILLSWITCH'),!.
'Fill'(Item):-
    update_menu_selection(Item,'Fill',
                [black,grey,lgrey,dots,check,crosses,diag,
                diamonds,horiz,rdiag,speckled,stripesthin,
                stripesthick,waves,white,alpha,beta],'FILL').
```

F.2.6

The **Round** menu generates a dialogue either to reset the round rectangle settings or the arc settings. The `rrectvals` and `arcvals` definitions are in the **dialogues** section of this program.

```
'Round'('Rectangle...'):-!,
    rrectvals,
    wfront(Name),
    inval_tool_pane(Name).

'Round'('Arc...'):-
    arcvals,
    wfront(Name),
    inval_tool_pane(Name).
```

F.2.7

This is a generalised routine for updating the markings on a menu.

```
update_menu_selection(Item,Menu,List,Property):-
    on(Item,List),!,
    get_prop('Graphic Editor',Property,Olditem),
    Olditem \= Item,
    set_prop('Graphic Editor',Property,Item),
    wfront(Name),
    inval_tool_pane(Name),
    remark(Menu,Olditem,Item).


remark(Menu,Item,Item):-!.
remark(Menu,Old,New):-
    unmark_item(Menu,Old),
    mark_item(Menu,New).
```

F.3 Graphic Editor : dialogues

The dialogues here make use of the advanced dialogue features of MacPROLOG (see the Advanced Dialogues chapter).

In each dialogue, integers must be entered, and we therefore associate with each dialogue a goal which checks that the user has typed integers (an 'Extended Dialogue'). The call to `mdialog` does not succeed until this checking program has succeeded.
If the user has not typed integers, a beep is sounded and a message is displayed.
(The user may of course click on the **Cancel** button in which case the `mdialog` call fails.)

Note the use of the `pname` primitive which converts a term into an atom which can then be displayed in a dialogue edit or text field.

F.3.1 The Round Rectangle dialogue

The current values of the oval width and oval depth for drawing a round cornered rectangle are stored as the `'OVALWIDTH'` and `'OVALDEPTH'` properties of `'Graphic Editor'`. These are retrieved for display in the dialogue, and they are then reset according to the user's input.

```
rrectvals:-
     get_prop('Graphic Editor', 'OVALWIDTH',OW),
     pname(OW,Ovw),
     get_prop('Graphic Editor', 'OVALDEPTH',OD),
     pname(OD,Ovd),
     mdialog(40,270,150,220,
            [button(120,20,20,60,'Ok'),
            button(120,140,20,60,'Cancel'),
            text(15,20,20,190,'Round Rectangle Corners'),
            text(45,30,32,100,'Oval depth:'),
            edit(45,140,16,35,Ovd,read(Depth)),
            text(85,30,32,100,'Oval width:'),
            edit(85,140,16,35,Ovw,read(Width))],
            Btn,intgs(Depth,Width)),
     set_prop('Graphic Editor', 'OVALWIDTH',Width),
     set_prop('Graphic Editor', 'OVALDEPTH',Depth).
```

The program to check that positive integers have been entered. This must fail if the integer checks fail, so that control is returned to the dialogue.

```
intgs(D,B,Depth,Width):-
     integer(Depth),
     integer(Width),
     Depth >= 0,
     Width >= 0,!.

intgs(D,B,Depth,Width):-
     beep(10),
     message('Please enter positive integers
                 for the~Moval depth and width'),
     fail.
```

F.3.2 The Arc details dialogue

The current values of the start angle and sweep angle for drawing an arc are stored as the `'START'` and `'ANGLE'` properties of `'Graphic Editor'`. These are retrieved for display in the dialogue, and they are reset according to the user's input to the dialogue.

```
arcvals:-
    get_prop('Graphic Editor', 'START',STA),
    pname(STA,SA),
    get_prop('Graphic Editor', 'ANGLE',AAA),
    pname(AAA,AA),
    mdialog(40,270,150,220,
          [button(120,20,20,60,'Ok'),
          button(120,140,20,60,'Cancel'),
          text(15,70,20,110,'Arc Details'),
          text(45,30,32,100,'Start angle:'),
          edit(45,140,16,35,SA,read(Sangle)),
          text(85,30,32,100,'Arc angle:'),
          edit(85,140,16,35,AA,read(Aangle))],
          Btn,intags(Sangle,Aangle)),
    set_prop('Graphic Editor', 'START',Sangle),
    set_prop('Graphic Editor', 'ANGLE',Aangle).
```

Check that integers have been entered for the angles. This is similar to the program for the Round Rectangle dialogue, except that negative integers are acceptable here, and a slightly different message is given for non-integer input!

```
intags(D,B,S,A):-
    integer(S),
    integer(A),!.

intags(D,B,S,A):-
    beep(10),
    message('Please enter integers for the angles'),
    fail.
```

F.3.3 The Pen details dialogue

The `curpvals` call picks up the current size of the pen. This is stored as the `'PENSIZE'` property of `'Graphic Editor'`, and may either be an atom such as `thick` or `thin`, or a term of the form `pensize(Depth, Width)`. Some term manipulation is necessary to cope with these two possible formats for the pen size.

```
penvals:-
     curpvals(Pnd,Pnw),
     mdialog(40,270,150,220,
           [button(120,20,20,60,'Ok'),
           button(120,140,20,60,'Cancel'),
           text(15,80,20,100,'Pen Size'),
           text(45,30,32,100,'Pen depth:'),
           edit(45,140,16,35,Pnd,read(Depth)),
           text(85,30,32,100,'Pen width:'),
           edit(85,140,16,35,Pnw,read(Width))],
           Btn,pintgs(Depth,Width)),
     set_prop('Graphic Editor', 'PENSIZE',pensize(Depth, Width)).
```

Check that positive integers have been entered for the pen details.

```
pintgs(D,B,Depth,Width):-
     integer(Depth),
     integer(Width),
     Depth >= 0,
     Width >= 0,!.

pintgs(D,B,Depth,Width):-
     beep(10),
     message('Please enter positive integers
                for the~Mpen depth and width'),
     fail.
```

Find the current pen values for the dialogue. The pen size may be an atom, or a term of the form `pensize(PD,PW)`, where `PD` is the pixel depth and `PW` is the pixel width of the pen.

```
curpvals(D,W):-
     get_prop('Graphic Editor', 'PENSIZE',pensize(PD,PW)),!,
     pname(PD,D),
     pname(PW,W).

curpvals(D,W):-
     get_prop('Graphic Editor', 'PENSIZE', Size),
     unmark_item('Pen',Size),
     mark_item('Pen','Pen size...'),
     sighs(Size,D,W).
```

Convert a pen size atom to actual size. (A `thin` pen is 1x1 pixels, a `thick` pen is 8x8 pixels, and a `nilpen` has zero size.)

```
sighs(thin,'1','1').
sighs(thick,'8','8').
sighs(nilpen,'0','0').
```

F.4 Graphic Editor : tools

These are the tool program definitions for the Rectangle, Round Rectangle, Oval and Arc tools.

Each tool has a `double`, `activate`, `deactivate` and 'click' mode defined.

The `double` mode is called when the user double clicks on the tool's picture. Here we give information on the tool using the `help` primitive.

The `activate` mode is called when the tool is selected, and the `deactivate` mode is called when the tool is deselected. In each of these cases we simply change the appearance of the graphic cursor.

The click mode is called when the user clicks in the viewing pane of the window; in this case the appropriate graphical object is drawn (the `drawing_tool` program is defined later).

For rectangles, we use the `'RBOX'` property of `'Graphic Editor'` to signal whether a round cornered or ordinary rectangle is to be drawn.

F.4.1 Rectangle Tool

```
geRect(double,Window):-
    tool_info('Rect').

geRect(activate,Window):-
    gcursor(Window,pen).

geRect(deactivate,Window):-
    gcursor(Window,cross_hair).

geRect(Window,Y,X,Mod):-
    drawing_tool(box,Window,Y,X,Mod).
```

F.4.2 Round Rectangle Tool

```
geRRect(double,Window):-
    tool_info('RRect').

geRRect(activate,Window):-
    set_prop('Graphic Editor','RBOX',on),
    gcursor(Window,pen).

geRRect(deactivate,Window):-
    set_prop('Graphic Editor','RBOX',off),
    gcursor(Window,cross_hair).

geRRect(Window,Y,X,Mod):-
    drawing_tool(box,Window,Y,X,Mod).
```

F.4.3 Oval Tool

```
geOval(double,Window):-
    tool_info('Oval').

geOval(activate,Window):-
    gcursor(Window,pen).

geOval(deactivate,Window):-
    gcursor(Window,cross_hair).

geOval(Window,Y,X,Mod):-
    drawing_tool(oval,Window,Y,X,Mod).
```

F.4.4 Arc Tool

```
geArc(double,Window):-
    tool_info('Arc').

geArc(activate,Window):-
    gcursor(Window,pen).

geArc(deactivate,Window):-
    gcursor(Window,cross_hair).

geArc(Window,Y,X,Mod):-
    drawing_tool(arc,Window,Y,X,Mod).
```

F.4.5

Generalised drawing routine for tools. We use the `marqui` primitive to establish the rectangle in which the graphical object is to be drawn. This `Frame` argument is then passed to `shape_args` which generates the appropriate graphical description of the object (see later). The definition of `myadd_pic` is in the **misc** window.

```
drawing_tool(Name,Window,Y,X,Mod):-
    marqui(Window,(Y,X),Frame),
    get_pen_details(Name,Syname,Size,
                        Penpattern,Fillswitch,Pattern),
    on(Name(Fillswitch,Prim),
            [Any(hollow,Any),box(filled,fillbox),
            oval(filled,filloval),
            arc(filled,wedge), poly(filled,fillpoly)]),!,
    shape_args(Name,Prim,Frame,Primandargs),
    myadd_pic(Window,Syname,
            Size(Penpattern(Pattern(Primandargs)))).
```

F.4.6

Give information on tools using the `help` primitive.

```
tool_info(Tool):-
    help('Editor Tools Help',Tool,'').
```

F.5 Graphic Editor : rubber band

These are the tool definition programs for the Lines and Poly tools. We use the `rubber_band` primitive (see `stretch_a_line`, defined later) to draw the lines for these tools.

F.5.1 Lines tool

```
geLines(double,Window):-
    tool_info('Lines').

geLines(activate,Window):-
    gcursor(Window,pen).

geLines(deactivate,Window):-
    gcursor(Window,cross_hair).

geLines(Window,Yy0,Xx0,Mod):-
    Initpt = (Yy0,Xx0),
    get_prop('Graphic Editor','PENSIZE',Size),
    get_prop('Graphic Editor','PENPAT',Ppat),
    actual_size(Size,Sizebox),
    actual_ppat(Ppat,Pattern),
    Pen = pen(Sizebox,Pattern),
    rubber_band(Window,Initpt,Next,Pen),
    stretch_a_line(Window,Next,[Next,Initpt],Pts,Mod,Pen),
    gensym(userLines,Syname),
    myadd_pic(Window,Syname, Size(Ppat(lines(Pts))))).
```

F.5.2 Poly tool

```
gePoly(activate,Window):-
    gcursor(Window,pen).

gePoly(deactivate,Window):-
    gcursor(Window,cross_hair).

gePoly(double,Window):-
    tool_info('Poly').

gePoly(Window,Yy0,Xx0,Mod):-
    Initpt = (Yy0,Xx0),
    get_pen_details(Size,Ppat,Fillsw,Fillpat),
    actual_size(Size,Sizebox),
    actual_ppat(Ppat,Pattern),
    Pen = pen(Sizebox,Pattern),
    rubber_band(Window,Initpt,Next,Pen),
    stretch_a_line(Window,Next,[Next,Initpt],Pts,Mod,Pen),
    gensym(userPoly,Syname),
    on(Fillsw-Prim, [hollow-poly,filled-fillpoly]),
    myadd_pic(Window,Syname,Size(Ppat(Fillpat(Prim(Pts)))))).
```

F.5.3
This program adds another line to the lines drawn so far, by waiting for a mouse click and calling the `rubber_band` primitive.
The second clause is to terminate the drawing when a 'modifier' key (e.g.*Shift*) is held down when clicking.
The third clause will be called if the user releases the mouse outside the window's viewing pane (this causes `rubber_band` to fail). In this case the `inval_box` has the effect of erasing all the (temporary) drawing done so far.

```
stretch_a_line(Window,Ptin,Sofar,Final,0,Pen):-
     wait_click(Window,Ny,Nx,Mod),
     rubber_band(Window,Ptin,Ptout,Pen),!,
     stretch_a_line(Window,Ptout,[Ptout|Sofar],Final,Mod,Pen).

stretch_a_line(Window,Last,List,List,M,Pen):-
     M \= 0,!.

stretch_a_line(Window,A1,A2,A3,A4,A5):-              /* any args */
     gview_pane(Window,Box),
     inval_box(Window,Box),
      fail.
```

F.5.4
Convert a pen pattern to a hexadecimal atom, and convert a pensize to a "point".

```
actual_ppat(greypen,'55AA55AA55AA55AA').
actual_ppat(blackpen,'FFFFFFFFFFFFFFFF').


actual_size(thin,box(0,0,1,1)).
actual_size(nilpen,box(0,0,0,0)).
actual_size(thick,box(0,0,8,8)).
actual_size(pensize(D,W),box(0,0,D,W)).
```

F.6 Graphic Editor : misc

This is a collection of miscellaneous routines used in various parts of the program.

F.6.1
Pick up the round cornered rectangle and the arc properties.

```
box_details(Depth,Width):-
        get_prop('Graphic Editor','OVALDEPTH',Depth),
        get_prop('Graphic Editor','OVALWIDTH',Width).

arc_details(S,A):-
        get_prop('Graphic Editor','START',S),
        get_prop('Graphic Editor','ANGLE',A).
```

F.6.2
The six argument form of `get_pen_details` returns the pen's various attributes as well as a suitable picture name (generated by `gensym`).
The five argument form just returns the pen's current attributes.

```
get_pen_details(Name,Syname,Size,Penpat,Fills,Fpatt):-!,
    gensym(Name,Syname),
    get_pen_details(Size,Penpat,Fills,Fpatt).

get_pen_details(Size,Penpattern,Fillswitch,Pattern):-
    get_prop('Graphic Editor','PENSIZE',Size),
    get_prop('Graphic Editor','PENPAT',Penpattern),
    get_prop('Graphic Editor','FILLSWITCH',Fillswitch),
    get_prop('Graphic Editor','FILL',Pattern).
```

F.6.3
The `shape_args` program returns a GDL description given a picture frame and the name of the graphical object. For a `box` object we need to add extra arguments if it a round cornered box. Similarly, the start and sweep angles must be added to an `arc` description.

```
shape_args(box,Prim,box(T,L,D,W),Prim(T,L,D,W,Hgt,Wdth)):-
    get_prop('Graphic Editor','RBOX',Flag),
    Flag = on,!,
    box_details(Hgt,Wdth).

shape_args(arc,Prim,box(T,L,D,W),Prim(T,L,D,W,Start,Amount)):-!,
    arc_details(Start,Amount).

shape_args(Other,Prim,box(T,L,D,W),Prim(T,L,D,W)).
```

F.6.4
This is a special form of `add_pic` needed by the Graphic Editor. Because the pen size may be in one of two forms, we cannot call `add_pic` directly.

```
myadd_pic(Window,Syname,Size(Penpat(Pat(Primandargs)))):-
    Size = pensize(A,B),!,
    add_pic(Window,Syname,pensize(A,B,Penpat(Pat(Primandargs)))).

myadd_pic(Window,Syname,Size(Penpat(Pat(Primandargs)))):-
    add_pic(Window,Syname,Size(Penpat(Pat(Primandargs)))).
```

LPA MacPROLOG™ Reference Manual

INDEX

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Transfer of the Software Package and End-User Licence

In accordance with the terms of the End-User Licensing Agreement granted by LPA to the original purchaser of the MacPROLOG 2.0 Software Package, the owner (either the original purchaser or a person to whom the Package and End-User Licensing Agreement have been transfered in accordance with the Agreement) may transfer the whole of the Package and the single-computer licence provided that the owner and the transferee both sign this re-registration form and agreement and return it within twenty-eight days to

Logic Programming Associates Ltd
Studio 4
The Royal Victoria Patriotic Building
Trinity Road
LONDON SW18 3SX

Please note that any offers which LPA may make of updates at special rates are available only to the Registered Owners.

Re-Registration Form and Agreement

Agreement between the Owner and Transferee and Logic Programming Associates Limited (LPA)

1. The Owner transfers to the Transferee the whole of the Package and all copies, updates and modifications to it (other than which have already been destroyed by the Owner) together with the Owner's licence to use the Package on a single-computer.

2. The Transferee agrees with LPA to accept all the terms and conditions of the End-User Licence granted to the original purchaser.

3. Both the Owner and the Transferee warrant to LPA that there is attached to every copy and modification of the Software a permanent label giving the name of the Software and stating that it is the property of LPA.

Signed by the Owner:

Name (please print) ....

Address (please print) ....

....

....

Signed by the Transferee:

Name (please print) ....

Address (please print) ....

....

....

Date: